KUNGL
TEKNISKA
HÖGSKOLAN

STOCKHOLMS
UNIVERSITET

Dynamic Projective Geometry

Harald Winroth

Dissertation, March 1999
Computational Vision and Active Perception Laboratory (CVAP)

Akademisk avhandling för
teknisk doktorsexamen vid
Kungl Tekniska Högskolan

Mars 1999

NADA, KTH, 100 44 Stockholm

ISRN KTH/NA/R--99/01--SE
ISSN 0348-2953: TRITA-NA-99/01
ISBN 91-7170-375-6

KTH Högskoletryckeriet
Stockholm 1999

Abstract

The theme of this thesis is *dynamic geometry*, a new way of exploring classical geometry using interactive computer software. This kind of software allows the user to make geometric constructions on a computer's screen. The constructions might consist of points, lines and conics whose positions have been constrained in various ways. The constraints, which may involve incidences, distances and angles, can be added and removed dynamically. For example, to force a line to always be incident on a point, the user would simply grab the line with the cursor and drop it onto the point. Any object whose position is not completely determined by the constraints can be grabbed and dragged around on the screen. The rest of the objects will then automatically self-adjust in order to keep the constraints satisfied. Dynamic geometry software is primarily used for teaching mathematics, but is useful in any situation where it is important to understand the geometric properties of a dynamic system.

Over the last few years, a number of tools for dynamic geometry have been developed. Most of them have focused on elementary Euclidean geometry. In this thesis we present a new software that has been based entirely on *projective* concepts and thus allows us to illustrate the classical theorems of projective geometry. The software has also extensive support for different types of *metrics*, which makes it possible to explore both Euclidean and non-Euclidean geometry. In fact, the user is given direct access to the *absolute elements* which define the metric. Moreover, the system can handle objects in the *complex projective plane*, which permits, for example, the *circular points* in Euclidean geometry to be used in geometric constructions.

We discuss how the user interface of a dynamic geometry system should be designed and we identify a number of problems and shortcomings which the user interfaces of all previous systems seem to suffer from. Most of these defects are related to the fundamental problem of choosing the "right" solution of an underdetermined system of constraint equations. We show how this problems can be solved by letting the system automatically add extra constraints if necessary, and by using a richer internal representation based on *oriented* projective geometry.

The thesis is written in English.

Keywords: dynamic geometry, visualization, computer-assistance in education, learning systems, constraint programming, user interface, projective geometry, oriented geometry, Euclid, metric.

Acknowledgments

I would like to express my gratitude to my supervisor Jan-Olof Eklundh for his constant support and encouragement over the years, and for always somehow finding financial support for me. I am equally grateful to my assistant supervisor Ambjörn Naeve. Without him, this thesis would never have been written. Ambjörn introduced me to the beautiful world of classical geometry and he has always been ready to discuss the problems that I have encountered. Ambjörn has also worked hard to promote the result of my work and he has carefully proof-read the entire thesis on short notice.

I would also like to thank Lars Svensson for explaining many concepts in linear algebra and geometry to me. I had very stimulating discussion with Jürgen Richter-Gebert when he visited KTH in 1995, and with Björn Sjödin during his short engagement here. Jiarong Li introduced me to the field of constraint programming and provided me with numerous references. I have learned much about software engineering and object-oriented techniques from several colleagues, in particular Matti Rendahl, Daniel Fagerström, and Björn Eiderbäck. Thanks to Tony Lindeberg and Lars Olsson with whom I have co-authored a few papers. Tony and I joined the group at the same time and he has ever since been a constant source of encouragement.

Thanks also to Carsten Bräutigam for helping me typesetting this manuscript in LaTeX, and to Birgit Ekberg-Eriksson and Emma Gniuli for helping me with administrative matters.

Finally, I would like to thank my beloved Victorine for her encouragement and endless patience, and for taking time out of her own research to help me finish my thesis.

Contents

Chapter 1

Introduction

1.1 Projective geometry and its history

The desire to study geometrical relationships arises in many fields of human activity, ranging from architecture to particle physics. To the early Greeks, geometry offered a way of describing the geography mathematically, which helped them make maps and navigate. Actually, the term "geometry" is derived from the Greek words for "earth measure".

Greek geometers were primarily interested in simple and very concrete geometric concepts, such as points, lines, triangles, ellipses, distances and angles. Soon, however, geometers came to realize that concepts line "point" and "line", which are very concrete and intuitive to us, could in fact be treated as abstract mathematical objects with no inherent meaning at all. Instead, the objects get their meaning only from their *relationship* to each other. Statements like "two lines intersect in a point", or "there is only one line that goes through two specific points" say something about how two types of objects, "points" and "lines", are related but nothing about what "points" or "lines" really *are*. This insight lead to the development of geometry as an *axiomatic system* and the writing of one of most influential geometry books ever written, Euclid's Elementa. Elementa described the geometry that are still taught today in high-school, *Euclidean geometry*.

In Elementa, Euclid developed his geometric theory from five postulates. The last one states that

> *If a line m intersects two lines k and l such that the sum of the interior angles on the same side of m is less than two right angles, then the lines k and l intersect on the side of m in which the sum of the interior angles is less than two right angles.*

(See Figure 1.1.) Apparently, Euclid felt uneasy about this postulate, because he tried to avoid using it in his proofs. Still, he was not able to derive the fifth postulate from the first four. Over the next thousand years, numerous attempts to prove the fifth postulate as a theorem were made. It was not until the 19th

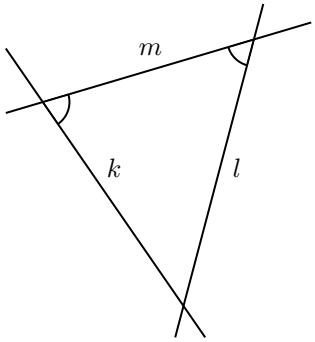


Figure 1.1. Euclid's fifth postulate.

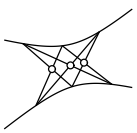
century that geometries in which the fifth postulate does not hold were discovered, and it became apparent that the fifth postulate was indeed independent of the other four. Two of the new geometries, *elliptic* and *hyperbolic* geometry, will be discussed further in Sections 4.12 and 6.4.

In the late 19th and early 20th centuries, the theory was generalized to higher dimensions. Coordinates were introduced and algebra became the primary tool for studying geometry. The theory of geometric transformations and groups was developed. Felix Klein was the first to consider any geometry to be a study of the properties that are invariant under a group of transformations, and he classified geometries using invariance as the criterion. For example, the group of transformations that leaves a specific line, the *line at infinity*, invariant constitutes *affine geometry* (see Section 4.12.4). Euclidean transformations were defined as the subgroup of affine transformations that preserves a certain distance and angle measure.

It was then obvious from group theory that all geometries were in fact special cases of a general geometry, defined by the full matrix group of transformations. This geometry was called *projective geometry*, and is the most general of all geometries, or as the great geometer Arthur Cayley put it: "projective geometry is all geometry".

One could say that projective geometry is the simplest of all geometries because it contains no special cases. In Euclidean geometry, for example, two lines meet in a common point *unless* they are parallel. In projective geometry, two lines always have a common point; there are no such things as parallel lines. The study of projective geometry is valuable because it tells us which geometric facts are true in general, and which properties are valid only in special geometries.

In this thesis, we will be concerned primarily with projective geometry, but will also look at Euclidean and non-Euclidean geometry, in particular the notion of metrics. However, we will restrict ourselves to the geometry of points, lines and conic sections (i.e., quadratic curves) in the plane.



1.2 Applications of projective geometry

Projective geometry is an important mathematical tool in many applied sciences, in particular Computer Graphics and Computer Vision. An important Computer Graphics problem is *rendering*, where a photo-realistic image is produced from the mathematical model of a three-dimensional object. The inverse problem, to identify a 3D object from a 2D image of it, is a typical Computer Vision problem. In both cases, it is important to describe algebraically the correspondence between 3D points in the real world and 2D points in the image. This turns out to be difficult in Euclidean geometry because of the perspective involved in the imaging process. For example, assuming that the imaging process can be described by a *pinhole camera model*, two parallel 3D lines will not appear to be parallel in image. In fact, their 2D line images will intersect in an image point, called the *vanishing point*, see Figure 1.2. Since two parallel lines do not intersect in Euclidean geometry, there is no 3D counterpart to the vanishing point, and thus, it is not possible to establish a one-to-one correspondence between objects in the image and objects in the world. One can also say that the effect of the perspective cannot be described in Euclidean geometry since Euclidean transformations by definition preserve parallelism. However, perspective effects can be easily expressed in projective geometry. In that geometry, two coplanar 3D lines always intersect in point. If the lines happen to be parallel in a Euclidean sense, they intersect at a *point at infinity*, which corresponds to the vanishing point in the image.

In recent years, there has been a growing interest in identifying and using image features that are *projectively invariant* to solve the recognition problem in Computer Vision. Because such features are not affected by the imaging process, we can expect to find them in pictures taken from any angle or distance. From the discussion above it is clear that parallelism is *not* an invariant feature, and consequently, it is no use looking for parallel lines in the picture even if there are parallel lines (edges or surface marks) in the scene. Distances and angles in the image also depend on the camera parameters and the position and orientation of the camera. In contrast, incidence relationships (e.g. that a point is on a line), tangencies, and cross-ratios (Section 4.3) are projectively invariant. Such features can be extracted from the image and used directly in the recognition process, for example as indices into a database of object models.

Because angles and distances are not preserved by the imaging process, there has been a tendency to disregard metric information altogether in recognition problems. That is not necessary; metric information is still useful in many situations. For example, suppose that a robot looks at an object that might be the tire of a car. The wheel can be described approximately by a circle with the wheel axis in the center – that is our metric model information. In a perspective image the wheel will appear elliptic and the axis will not be at the center of that ellipse, see Figure 1.3. Can the robot use this knowledge to verify that it is looking at something that could be wheel? Imagine that we draw all diameters of the wheel in 3D and for each diameter look at the 3D tangents in the two points where the

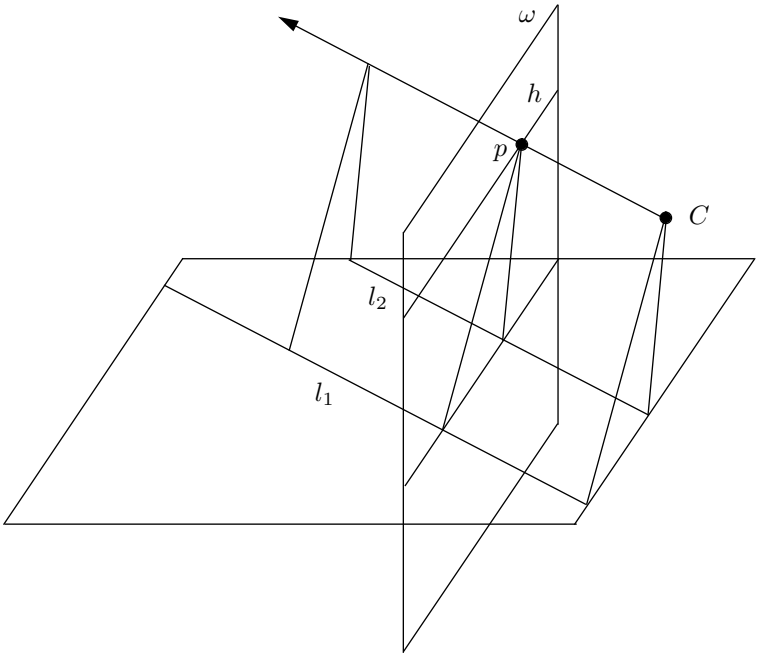


Figure 1.2. Two parallel lines l_1 and l_2 do not intersect in Euclidean three-space, but their two-dimensional images in the image plane ω do. C is the center of projection, h is the horizon, and p is the vanishing point.

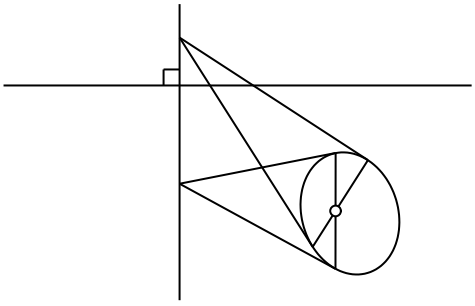
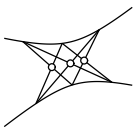


Figure 1.3. A perspective view of a tire in a vertical plane. The optical axis of the camera is parallel to the ground plane.



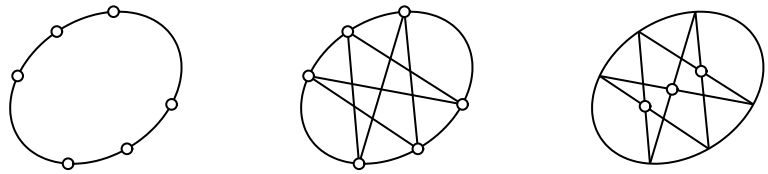
diameter meets the circle. Each pair of tangents will be parallel in 3D and all tangents will be in the plane of the circle. The points in which the 2D images of the tangent line pairs intersect should therefore be collinear if the wheel hypothesis is correct. This is easy for the robot to verify. The intersection points will be on the horizon of the plane of the wheel. Furthermore, if the plane of the wheel is perpendicular to the ground plane (a reasonable assumption for a the wheel of a car), and the optical axis of the camera is parallel to the ground plane, the horizon of the wheel plane should be perpendicular to the horizon of the ground plane in the image. That can also easy to check.

In this thesis, we will primarily deal with projective geometry and its theorems, but because of the growing interest in using metric information in a projective setting, we will discuss metrics from a projective point of view. Instead of just introducing the formulas for the distance between two points or the angle between two lines that are peculiar to a certain type of geometry, we will consider the metric introduced by a quadratic form in the surrounding projective space. Then, an angle between two lines can be represented by the cross-ratio of four lines and the distance between two points can be represented by the cross-ratio of four points (Section 4.12.3).

1.3 Dynamic geometry for visualization

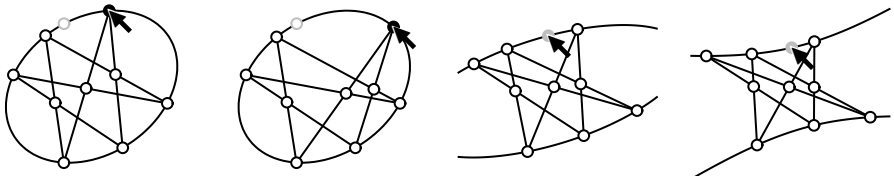
In this thesis we discuss *dynamic geometry software* as a tool for geometric visualization. Why are visualization tools at all important in such a well-understood field of mathematics as projective geometry, especially when there exists powerful algebraic methods that are not limited to problems in two or three dimensions? Although it is true that the theory of projective geometry is well-developed, the sheer size of the field makes it very difficult for students and researchers to get an overview. There is an enormous amount of known facts that has to be grasped; text books usually only covers the most important theorems, and only occasionally illustrates them with simple sketches. In general, it is not difficult to follow an algebraic proof of a theorem. The difficulty is to know what theorems to apply when dealing with a particular application. That is what visualization techniques, and in particular dynamic geometry systems are all about: to expose principles and known facts in a way that can give the user new insights and a better understanding of the theory and how to apply it. We argue that in this role, geometric visualization cannot be replaced by algebraic methods. However, once a phenomenon (e.g. an incidence relation or an invariant) has been discovered through experimentation and visualization techniques, it is in general a minor problem to prove it using algebraic methods. Actually, a large class of theorems in projective geometry can be proven automatically [Kutzler86, Wu86, Richter-Gebert95]. Thus, geometric visualization does not exclude the use of algebraic methods or vice versa.

To make this discussion more concrete, let us look at an example. The classical theorem of Pascal states the following: if we select six arbitrary points on a conic



(a) (b) (c)

Figure 1.4. The theorem of Pascal.



(a) (b) (c) (d)

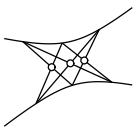
Figure 1.5. Pascal's theorem as a dynamic drawing.

(Figure 1.4a), then draw the six lines shown in Figure 1.4b, the three intersection points shown in Figure 1.4c will then always be collinear. This property is not self-evident; it requires a proof. Unfortunately, the algebraic proof will not really help the reader to understand intuitively what is going on and why the theorem is true. However, if Figure 1.4c had been a *dynamic sketch*, the reader could have dragged the points on the conic or changed the shape of the conic and watched what had happened. Dynamic drawings can only be fully appreciated on a screen, so the reader is encourage to try this example on a computer. However, the sequence of snapshots in Figure 1.5 shows the general idea.

A dynamic drawing consists of a limited set of elements such as points, lines and conics. In contrast to a standard drawing editor, these geometric elements can be *constrained* so that, for example, a certain point is always on a certain line or conic. Compared to compass-and-ruler drawings on paper, dynamic geometry software offers a number of advantages:

- Drawings will be *accurate*.
- *More than one case* can be shown.
- *Invariant* features can be visualized.
- *Duality* can easily be explored.
- *Exploration* and *experimentation* are encouraged.

Let us look more closely at each of these claims.



Inaccurate, free-hand paper drawings are dangerous, because we easily make false assumptions when we see them. A classical case is a "theorem" which state that all triangles are isoceles [King97]. A "proof" of this theorem can convincingly be derived from an inaccurate drawing, such as the one shown in Figure 1.6.

> ABC is a given, arbitrary triangle. Draw l, the angle bisector of A, and m, the perpendicular bisector of BC. l and m intersect in a point D. The perpendiculars from D to AB and AC intersect the triangle at F and G, respectively. Draw BD and CD (see Figure 1.6).
>
> Because of symmetry, $ADF \cong ADG$ and $EDB \cong EDC$. Therefore, $DF = DG$ and $DB = DC$. Thus, $DFB \cong DGC$ and $FB = GC$. Since also $AF = AG$, we have $AB = AF + FB = AG + GC = AC$.

The problem with Figure 1.6 is that points F and G are not really on the same side of the base line BC. A series of correct drawings is shown in Figure 1.7. From these drawings, it is obvious that $AF + FB = AG + GC$ is false and that the proof is invalid. It has been stated that it is better to make no drawings at all rather than making inaccurate ones. We agree with that, but argue that *accurate* drawings created by software are an invaluable aid when searching for an algebraic proof.

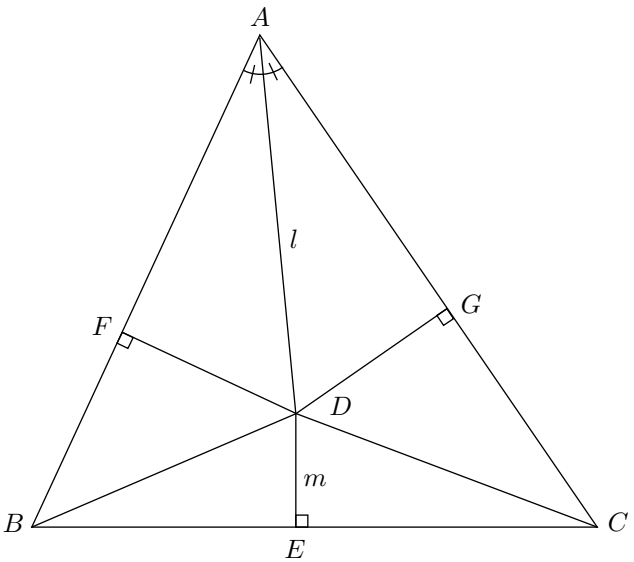


Figure 1.6. An illustration of the "theorem" that all triangles are isoceles.

Making a hand-drawn set of figures which illustrates various cases of a geometric configuration is far too time consuming, especially if the drawing is complicated and contains curves. Consequently, theorems in geometry text books are either not illustrated at all, or illustrated with a single sketch. Within reach is now

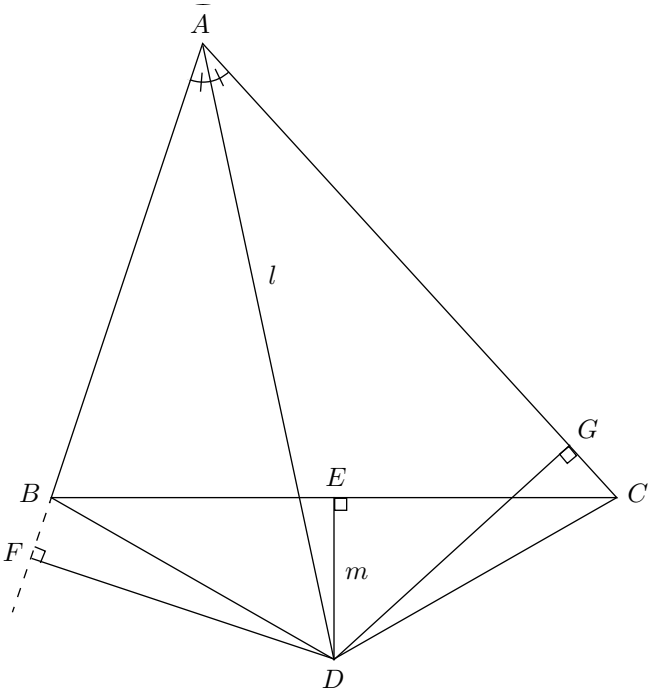


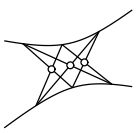
Figure 1.7. An accurate drawing of the "isoceles" triangle.

the possibility of creating multi-media text books which illustrate theorems with modifiable sketches and animations. Even when producing paper books, such as this thesis, a rich set of illustrations can be created with little effort using dynamic software.

Drawings on paper are always static. Since nothing can move is such drawings, it is not possible to see what *does not* change – the invariants. Consider the drawing in Figure 1.8a. P, Q and R are three arbitrary points on a circle. l is the bisector of PQ and PR. What will happen when point P is moved? From the static drawing, that is hard to see. In Figure 1.8b, a number of superimposed snapshots of l are shown. That figure clearly shows that the intersection point of l and the circle is invariant[1].

For every theorem in 2D which says something about incidences between points, lines and conics, there is a dual theorem where every "point" has been replaced by "line" and every "line" has been replaced "point". Since points and lines satisfy the same set of incidence axioms (see Sections 4.1 and 4.2). and since all theorems are ultimately derived from these axioms, any proof about a point configuration will also be valid for the corresponding line configuration. For example, the dual of Pascal's theorem, often called Brianchon's theorem, is illustrated in Figure 1.9. The conic has been inscribed in a hexagon consisting of six

[1] This can easily be proved. See Section 6.3.1.



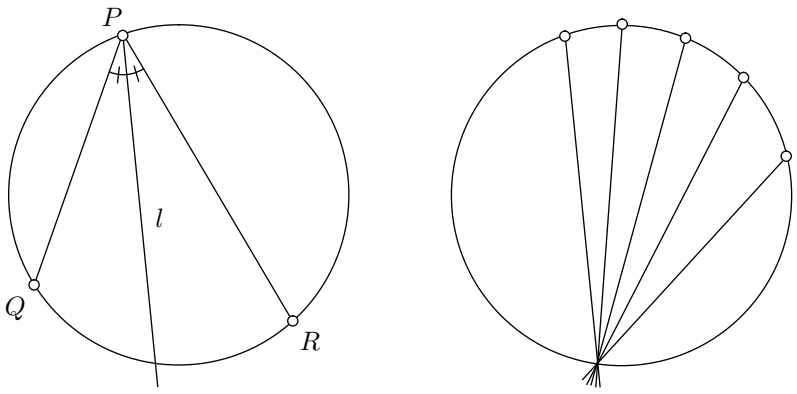


Figure 1.8. P, Q and R are three arbitrary points on the circle. l bisects the angle QPR. If P is dragged along the circle, what is invariant?

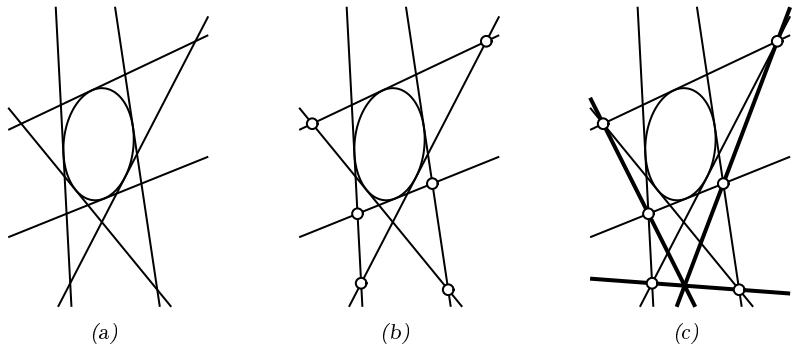
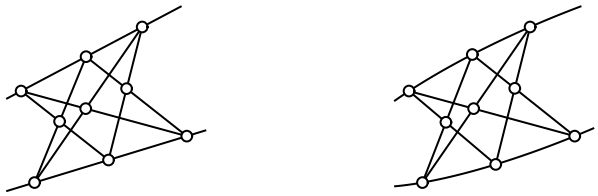


Figure 1.9. The theorem of Brianchon.

arbitrary tangents to the conic. The lines joining opposite vertices will always intersect in the same point. Although Figure 1.9 illustrates the same theorem as Figure 1.5, the drawings look very different. Interchanging point and lines are often called a *change of background*, and has a great pedagogical value, because it can make us see beyond a concrete representation of a geometry and better understand the underlying structure. Dynamic software can automatically create the dual of any drawing and visualize the correspondence between the elements in the two drawings.



(a) (b)

Figure 1.10. The theorems of Pappus (a) and of Pascal (b).

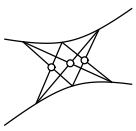
When looking at illustrations of theorems, a number of what-if questions often come to mind. For example, if a conic degenerates into a double line, is Pascal's theorem still valid? In Figure 1.10a, we have replaced the conic by two lines. The three intersection points still appear to be collinear. Although the sketch cannot provide any proof of this, this property is true in general (Pappus' theorem). We could have forced the conic in Figure 1.5 close to its degenerated shape, as shown in Figure 1.10b. It then becomes apparent that the theorem of Pappus is a special case of Pascal's theorem. This example illustrates the usefulness of dynamic geometry software for exploring and testing conjectures.

1.4 Who will the users be?

In what situations could a dynamic geometry system be useful? What kind of users do we have in mind, and what kind of problems will they study? In USA and France, dynamic geometry systems have been used extensively in high-school geometry teaching, with a focus on classical, two-dimensional Euclidean geometry. Some attempts to use these systems at college level have also been reported [Hamson97].

Dynamic geometry systems can also be used in advanced mathematical research to look for new theorems, and to prove automatically the theorems found [Richter-Gebert95]. Automatic theorem proving is a research field in itself and beyond the scope of this thesis. However, we shall discuss briefly the usefulness of these methods in Section 5.3.4.

We also expect that dynamic geometry systems will be used by researchers in applied sciences, such as Image Processing, Computer Vision and Robotics.



A dynamic geometry system makes it possible to simulate real-world situations with appropriate object and camera models. By operating controls on the screen, the user can modify all camera parameters, the viewpoint and the position of the objects. In this role, dynamic geometry systems could turn out to be invaluable tools for studying the imaging process. Actually, some groups have already produced course material based on dynamic geometry intended university level Computer Vision courses [Backus97].

When constructing the software presented here, we have attempted to provide more support for advanced geometric constructions than what is generally found in existing software. In particular, we have been interested in studying non-Euclidean geometries, metrics defined by projective elements (such as absolute conics), and computations in the complex projective plane. We hope that our software can be used by researchers in applied sciences and by students in courses given at university level.

1.5 Special purpose software or general software packages?

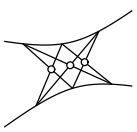
What kind of systems can be used for studying the concepts and theorems of classical projective geometry in two and three dimensions? Over the last decades, there has been a tremendous development of visualization tools, ranging from two- and three-dimensional CAD modelers to systems for symbolic computations and visualization such as Maple [Monagan98] and Mathematica [Wolfram96]. However, we will argue that none of these systems are well suited for direct manipulation of the type of geometric figures we are interested in.

Three-dimensional CAD systems are capable of showing an object in a perspective view from any angle. In order to produce perspective views, compute occlusions and intersections, CAD systems embody many concepts from projective geometry, such as homogeneous coordinates and projective transformations. However, these tools are primarily intended to create objects that are visually appealing, with smooth corners and surfaces, and not for mathematical studies of geometry. CAD systems are often oriented towards computations with spline patches, since splines allow a wide variety of shapes to be created, and at the same time makes rendering relatively easy. A CAD system has in general no deeper understanding of the underlying mathematical structure. In particular, there is usually no explicit representation of simple shapes such as cones or cylinders. A cylinder is typically made up by a set of surface patches, and if the radius of the cylinder tends to zero, the system will not realize that the cylinder has collapsed into a line. If the cylinder is intersected by a plane, a CAD system cannot in general tell that the intersection curve is a conic which can be represented by a particularly simple equation. As a consequence, it is not possible to directly use the features of the conic (such as its axis or its focal points) in other constructions. Many modelers are in fact are inconsistent and allow nonsense objects or empty sets [Requicha80]. Another problem with using CAD systems for investi-

gations of geometry is that a Euclidean embedding is often implicitly assumed. Hence, pure projective geometry, hyperbolic geometry and elliptic geometry must be simulated in a Euclidean setting. For the same reason, affine and projective distortions may be difficult to visualize.

Systems like Mathematica, on the other hand, support symbolic computations and can simplify and solve algebraic equations without resorting to numerical methods. Mathematica provides a language that is powerful enough to encode both simple geometric objects (points, lines, conics) as well as more complicated objects (three-dimensional algebraic curves and surfaces). However, symbolic computations are orders of magnitude too slow for the kind of interaction we have in mind. In addition, the user interface plays a central role in all dynamic geometry software and systems like Mathematica have limited support for implementing user interfaces.

These considerations have led us, and a few other research groups, to create special-purpose software which allows classic projective geometry to be studied interactively. We have tried to find an appropriate representation of projective geometry which makes the geometric knowledge explicit and at the same time allows geometric computations to be performed at high speed. This could be seen as an attempt to bridge the gap between the relatively slow, symbolic computational engines and the faster rendering systems which lacks geometric knowledge. In the next chapter, we will review a number of existing dynamic geometry systems.



Chapter 2

Related work

A number of dynamic geometry systems have been created over the last few years, and several of them are still being developed. The systems roughly fall into two categories, which could be called *constraint-based* and *constructive*. A user of a constraint-based system first creates a set of objects, such as points, lines and conics, and then restricts the position of the objects until only one or a finite number of geometric configurations are possible. Constraint-based systems usually employ a general constraint programming method to find a configuration which satisfies all the constraints specified by the user. In a constructive system, on the other hand, objects and constraints are indivisible. The user can create objects such as "tangent to a conic" or "midpoint of a segment", which represent both a geometric shape (here, a line or a point) and a constraint (here, the line and the conic must have exactly one point in common, or the distances between the end-points of the segment and the new point must be equal). Constructive systems encourage the user to think about sequential ruler-and-compass constructions, while users of constraint-based systems have to think about specifications and the number of possible solutions. In principle, constraint-based systems are more powerful and allow for a higher degree of interactivity. However, it turns out that these systems are also harder to use in many situations and significantly slower than the constructive ones. It seems that constraint-based systems can in practice only be used for very simple drawings. The distinction between the two types of systems will be discussed further in Section 5.1.

The system presented in this thesis (pdb) and three of the systems introduced below (Cabri II, GSP and Cinderella Café) are all constructive. Cabri II and GSP are by far the most widely used systems in education. Cinderella Café is not that well-known but has many interesting features. The constraint-based group is here represented by three systems, GéoSpécif, UniGéom and Juno-2. While there is a large number of constraint-programming systems that deal with automatic layout and CAD, GéoSpécif, UniGéom and Juno-2 have been written specifically for dynamic, projective geometry.

2.1 Cabri

Cabri [Baulac92, Schumann94] was originally developed by the EIAH team of Leibniz laboratory and the Institut d'Informatiqe et de Mathematiques Appliquees in Grenoble (IMAG) under the leadership of Jean-Marie Laborde. The first version was presented at the 6th International Congress of Mathematical Education in Budapest 1988. The latest release, Cabri II, is marketed by Texas Instruments, while research and development continues at IMAG.

Cabri II drawings are made up of points, lines, conics and polygons. Points can be attached to lines and conics and lines can be restricted to go through one or two points. Tangency is not a primitive constraint in Cabri II, and tangents to a conic on a given point must be constructed from other primitives, although such a construction can be created by a macro. Euclidean angles and distances can be measured and used in constraints.

The user interface is very good. Most incidence constraints can be defined using drag-and-drop operations, and useful feedback is given in most situations. However, defining metric constraints, in particular constraints on angles, is quite cumbersome.

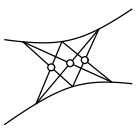
The position of objects are represented internally by homogeneous coordinates, but to the user, Cabri II appears to be heavily biased towards Euclidean geometry. For example, a Euclidean metric is implicitly used for measuring angles and distances, and the same metric is applied in all views (windows) of the drawing. This approach works only because the coordinate systems of different views are related by Euclidean isometries. It will not work in a system where each view can show a general, perspective transformation of the underlying drawing.

2.2 GSP (Geometer's Sketchpad)

GSP (Geometer's Sketchpad) [Jackiw95] originates from the Visual Geometry Project at Swarthmore College in the mid-80's. The original proposal for GSP placed the emphasis on solid geometry, but because of limited computational power available at the time, the system was restricted to planar geometry instead. GSP is now a commercial product marketed by Key Curriculum Press. However, research and development of the product is still supported by research grants from the National Science Foundation. The principal designer of GSP and the leader of the current development is Nick Jackiw at Key Curriculum Press.

GSP drawings may include points, lines and circles, but not general conics. Tangents to circles are not primitives, but can be constructed from other elements by a macro. Measurements and transportation of angles and distances are supported. GSP is intended for investigations into Euclidean geometry and therefore has no provisions for general perspective transformations.

The user interface is quite good although it is not as smooth as that of Cabri. However, GSP has a more advanced macro facility and allows custom tools to be created.



2.3 Cinderella Café

The Cinderella Café dynamic geometry system was developed primarily by Jürgen Richter-Gebert, Technischen Universität Berlin and ETH Zürich, in cooperation with Henry Crapo and Ulrich Kortenkamp. The first version was written for the NEXT system, but unfortunately, the program was never released and no documentation was published. The current version has been written entirely in Java and can therefore be run from any Java-capable web browser on any platform. A brief description of the system and a demo version is available on the Cinderella Café home pages [Richter-Gebert98]. A new version of Cinderella Café has been developed and will available in March 1999. The software and an accompanying booklet will be distributed by Springer-Verlag [Richter-Gebert99]. However, at the time of writing, the book and the new version of the program are not yet available from the publisher. Therefore, what is said here about Cinderella Café is based on information gathered through experimentation with the 1998 demo version.

Cinderella Café supports points, lines and conics, but not distance/angle measurements or metric constraints. Tangents to a conic that passes through a given point can be created, but not the common tangents of two conics. Geometric constructions can be dualized automatically, although this feature has not been fully implemented. A construction can also be displayed simultaneously as a Euclidean drawing, on a Poincaré disc, and on a sphere.

Cinderella Café has an advanced proof engine which can verify that a property observed by the user, for example that three points appear to be collinear, is true in general. The proof is produced by algebraic methods and not by a simple numerical test.

2.4 GéoSpécif

GéoSpécif [Allen93, Bouhineau95, Allen97] is a constraint-based system which automatically constructs a drawing from a set of logical constraints. The user first declares a set of geometrical objects, which can be points, lines and segments, then specifies how the objects are related. It is possible to specify that a point must be on a certain line or that two lines must be parallel or perpendicular. The distance between two points can also be specified, but not the angle between two lines.

The system, which has been implemented in Prolog, has a limited capability for solving non-linear equations. The strategy of the system is to solve the linear parts of the system first, then use the result to linearize the non-linear equations or rewrite the non-linear equations in a simple form which can be solved directly. Obviously, there are many specifications for which this strategy fails, for example the construction of a regular pentagon or the common tangents of two conics. However, it is possible, for example, to create the tangent from a given point to a circle, if the user provides the system with over-specifications.

The system updates the drawing automatically if the user drags one of the (unconstrained) base points. For constructions involving only linear equations, the response time is reported to be acceptable. For constructions resulting in non-linear equations, however, the response time may be several seconds.

2.5 UniGéom

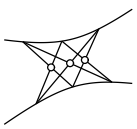
UniGéom [Channac96, Bouhineau96] is another constraint-based geometry program written in Prolog. It is similar to GéoSpécif but is slightly more powerful since it contains circles as primitive elements, and it is possible to specify directly that a line is tangent to a circle. However, general conics are not supported. UniGéom uses a strategy similar to that of GéoSpécif for solving non-linear equations: it solves the linear equations first and then simplifies the non-linear parts.

2.6 Juno-2

Juno-2 [Heydon94] is an experimental, constraint-based drawing editor. A Juno-2 drawing is typically produced in three steps. The user first draws an approximate sketch, and then restricts the position of the drawing primitives by specifying constraints in a declarative extension language. The editor will then automatically modify the sketch so that all constraints become satisfied. If the sketch is under-constrained, the editor will choose a configuration that is close to the original, approximate sketch. Finally, the user may adjust the position of control points until the result is satisfactory. When a drawing is adjusted, the editor makes sure that all constraints remain satisfied.

Juno-2 can handle non-linear constraints, and the set of possible constraints is not fixed; constraints can be declared using a subset of first-order logic. The resulting set of constraints are solved by a combination of symbolic manipulation and numerical methods.

Although Juno-2 seems to be intended primarily for graphical design, it has also been used for illustrating theorems in projective geometry. However, it does not have the speed and high interactivity that is characteristic of pure dynamic geometry systems.



Chapter 3

Aims and contributions

The idea of building a dynamic geometry system has been around at our lab since 1985. A simple prototype was completed in 1988 [Naeve89]. Although that system could only handle points and lines, it effectively demonstrated the basic idea. Other, more specialized systems were also being built at that time, for example to study the the progression of wave fronts in mirrors [Naeve93, Appelgren94]. Under the supervision of Ambjörn Naeve, the current author began to develop a more complete, robust and easy-to-use system for exploring plane projective geometry in 1995. The result of this work, the Projective Drawing Board (pdb), is presented in this thesis.

pdb has been developed in parallel with, and been inspired by, the work made by other research groups. New versions of Cabri II and Cinderella Café (see Chapter 2) have been released since the development of pdb began. Inevitable, there is some overlap between the systems, and they have many features in common. However, they were developed with different target groups in mind. Cabri and GSP were primarily intended for teaching Euclidean geometry at high-school level. Cinderella Café has a more mathematical flavor, with an emphasis on automated theorem proving. pdb was intended for teaching geometry at university level, and for applied research in Computer Graphics and Computer Vision. As a consequence, the mathematical kernel and the user interface that these systems offer differ significantly.

The contributions of pdb to the family of dynamic geometry systems falls into three categories:

- the design of the mathematical kernel, where both the expressive power and the speed of the calculations have been primary concerns,
- the significantly improved user interface, and
- the structured, open-ended and well documented system design.

A more detailed account will be given in the next three sections.

3.1 Mathematical kernel

We wanted pdb to be based on concepts from projective geometry, with no implicit assumptions about Euclidean metrics or of a particular embedding in R^3. We argue that metric support should be added on top of the basic projective model, just as the mathematical theory suggests (Section 5.2.7).

pdb allows angle and distance constraints to be expressed in any metric, and the resulting sketch can be displayed in any type of view, for example a plain Cartesian view or a Poincaré disc (Section 4.12.1). Both GSP and Cabri are heavily biased towards Euclidean geometry, and the users of these systems must apply various tricks in order to visualize other kinds of geometry [Schumann94]. Cinderella Café is the only existing system that seems to be truly projective. However, we feel that Cinderella Café does not have the same level of support as pdb has for studying and modifying different metrics.

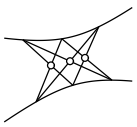
None of the other tools presently has enough expressive power to directly support the construction of several important drawings. For example, only Cinderella Café supports tangents to a conic as a geometric primitive, and only in a limited sense[1]. In pdb, we wanted support for one-parameter families of tangents of conics, tangents of conics through a point, and common tangents of two conics (Section 5.2.1).

The ability to place metric constraints on objects is vital when Euclidean, hyperbolic and elliptic geometries are studied. It should be easy to transport angles and distances from one part of a drawing to another. Cabri and GSP support angle and distance measurements, and a limited set of metric constraints. Compared to Cabri, however, pdb contributes an improved user interface design for placing metric constraints on objects, and it also has a richer set of geometric primitives involving metrics. The currently available version of Cinderella Café lacks support for measurements and for metric constraints.

In projective geometry, angles and distances have no meaning since they are not preserved by projective transformations. However, the cross-ratio of points and lines is invariant under general projective transformations, and constraints involving cross-ratios are therefore meaningful in all geometries. Surprisingly, no other tool seems to handle directly constraints involving cross-ratios. This again demonstrates the implicit assumption of a Euclidean geometry. pdb fully supports the notion of cross-ratios, and any point or line can be constrained using cross-ratio relationships.

The tools described in Chapter 2 deal only with the real projective plane, and have no support for managing complex coordinates. However, some objects with complex coordinates seem to appear naturally in studies of real projective geometry. For example, a real line intersects a conic (with a real point set) in two real points if the line contains an interior point of the conic, see Figure 3.1a. However, if the line is moved outside the conic, no real intersection points exist (Figure 3.1b). This is the kind of annoying special cases that are so familiar from

[1] Tangents to a conic through a point are always drawn as a degenerated conic (a pair of lines). Also, the tangent points cannot be constructed.



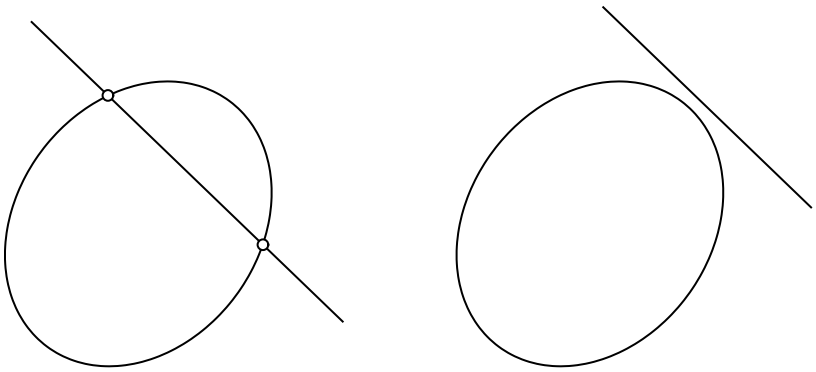


Figure 3.1. A line and a conic intersect in two real points only if the line contains an interior point of the conic.

Euclidean geometry where, for example, two lines intersect in one point, *except* when they are parallel. The number of special cases tends to grow exponentially with the number of objects in a drawing and we cannot afford to ignore them; there will always be some objects in the drawing that become undefined and make the construction collapse. All of this can be avoided if we allow for complex coordinates. The line in Figure 3.1b actually intersects the conic in two conjugate complex points, which are incident both with the line and with the conic. pdb is able to calculate and visualize shapes with complex coordinates when necessary, although the real solutions are preferred when they are available (Section 5.3.4). One of the goals with pdb was to enable the user to interact with complex objects.

pdb's ability to directly deal with complex coordinates also allows us to access and manipulate what is known as the *absolute elements* in a geometry. For example, the Euclidean angle measure can be defined in terms of a conjugate complex point pair, known as the *circular points*. In elliptic geometry, the metric is given by a conic, whose point equation has real coefficients but only complex roots (Section 4.12.2). This thesis contains several examples which demonstrates how the absolute elements can be used, see e.g. Sections 5.3.6, 6.3.2, 6.3.3, and 6.4.1.

3.2 User interface

pdb has a drag-and-drop interface for creating drawings which is similar to that of Cabri. In both systems, constraints are defined by dropping objects onto each other. However, pdb's drag-and-drop interface also allows all incidence constraints to be removed or redefined dynamically. In addition, the drag-and-drop interface has been extended to handle constraints involving distances, angles, and tangencies.

pdb tries to capture the intent of the user and to suggest suitable operations and operands in each given situation. For example, if the user drags a line onto a conic, pdb suggests that the line should be a tangent to the conic, if that is geometrically possible. Also, several objects can be created in one operation (Section 5.3.2). If the user drops a line onto the intersection of two other lines, pdb first creates the intersection point, and then restricts the dropped line to be incident with that point.

A common weakness in existing tools is the treatment of under-constrained objects, i.e. objects whose positions are partly but not completely determined by geometric constraints. Examples are a point that has been restricted to be on a certain line, and a line that must be tangent to a given conic. When a line is dragged, free points on that line often drift together, which usually cause a part of the drawing to collapse. It is just as common that the free points slide away along the line and disappear from the screen (Section 5.2.3). In pdb, under-constrained objects always move in a well-defined and predictable manner. When the position of an under-constrained object needs to be updated, additional geometrical constraints are automatically added by the system until the new position is completely determined. The extra constraints depend on the geometric structure of the drawing, the object being dragged, and the properties of the view in which the interaction takes place (Section 5.2.6).

Existing tools also have problems with under-constrained objects that can be placed in one of a *finite* number of positions. For example, a line and a conic intersect in two points. The user will often select one of these points and use it as a basis for the next step in the construction. Then, if the line or the conic is moved slightly, the selected intersection point often jumps unpredictably between the two positions it can occupy. To overcome this problem, pdb uses *oriented* projective geometry [Stolfi91] in all calculations, which makes it easier for the system to distinguish multiple roots of a geometric equation. The theory will be reviewed in Section 4.13 and the identification of roots will be discussed in Section 5.2.5.

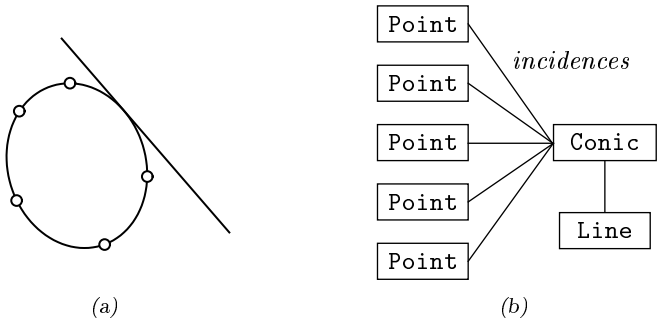
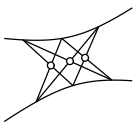


Figure 3.2. A conic on five points and one of its tangents.

When working with a complicated drawing it is often necessary to see the *logical structure*, i.e., how the objects depend on each other and which constraints



have been placed on the objects. pdb can display a drawing as a constraint graph whose nodes represent shapes and whose arcs represent constraints, see Figure 3.2. It is possible to interact directly with this graph and, for example, add and remove nodes.

3.3 System design

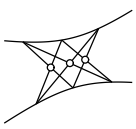
All of the systems described in Chapter 2 are either commercial or are about to become commercial. Therefore, the internal data structures, the algorithms and the source code are unavailable. Very little about design decisions, trade-offs and implementation details has been published. One goal of the present work is to make this information available, in the hope that it can serve as a starting point for other developers of dynamic geometric software. In Section 5.4 the design of pdb has been documented in UML, the Unified Modeling Language [Booch99].

The design of pdb is object-oriented with well-defined interfaces and object interactions. The system is open-ended: new geometrical primitives and constraints can be added, and existing interaction strategies may be replaced by new ones (Section 5.4.6).

Most dynamic geometry systems have some kind of built-in macro language which allows the user to specify a drawing as a program rather than drawing it interactively on the screen. Every system with this capability that we have seen has its own macro language with a non-standard syntax. In pdb, we have taken another approach and extended a standard interpreted language with geometric functions. Actually, a very general way of integrating the macro language with the compiled, strongly-typed implementation language has been developed and is presented in Appendix A (also in [Winroth98]).

Portability has been another major concern. All system-dependent code, in particular code relating to the windowing system, has been carefully isolated. Only standard libraries have been used for creating the internal data structures.

Even though the design of pdb is object-oriented, unnecessary layers and interfaces have been removed, and the use of resource-consuming windowing toolkits have been avoided. This has resulted in a compact and fast system, and we have been able to keep the response time of the system short, despite the amount of extra computations necessary to achieve the improvements of user interface mentioned above.



Chapter 4

Mathematical background

To make this thesis self-contained, we provide the necessary mathematical background in this chapter. It may be skipped by readers familiar with plane projective geometry. However, in subsequent chapters we will frequently refer to the definitions and theorems presented here.

There are two basic approaches to projective geometry. It can be seen as a logical system based on a set of axioms relating the undefined elements "point" and "line" (synthetic projective geometry), or it can be given an algebraic definition based on the notion of vector spaces (analytic projective geometry). We have chosen the latter approach since it not only makes the presentation shorter, but also provides a better framework for geometric computations.

It is our ambition to define clearly every concept we use and to prove (or at least outline a proof of) every theorem we mention. However, this chapter is not intended to be used a text book on projective geometry. We will mostly restrict ourselves to concepts and theorems that are fundamental to the user interface and implementation of pdb. In addition, a few theorems that will be used in the examples will also be proved. We will be concerned only with points, lines and conics in the projective plane, not with the projective spaces of higher dimensions or general algebraic curves or surfaces. For a more thorough presentation and a broader mathematical perspective on the subject, we refer the reader to [Klein25, Klein28, Salmon60, Winger62, Coxeter93, Coxeter98, Samuel88, Meserve59, Bix98].

We have tried to keep formal derivations to a minimum and to avoid heavy mathematical machinery. However, since our approach is algebraic the reader is assumed to be familiar with the basic concepts from linear algebra such as vector spaces, metrics, determinants, and eigenvalues.

4.1 Point and lines in the projective plane

In elementary Euclidean geometry, a point p is represented by its coordinates (x, y) in some coordinate system S, i.e., (x, y) is the position of p relative to the

origin of S. To emphasize that the coordinates depend on the choice of coordinate system, we sometimes write $(x, y)_S$. However, we can instead say that the pair of numbers (x, y) itself *is* the point, and that the set of points in the Euclidean plane is $\{(x, y)\colon x, y \in \mathrm{R}\}$. That allows us to speak about *the point* (x, y) without referring to any coordinate system.

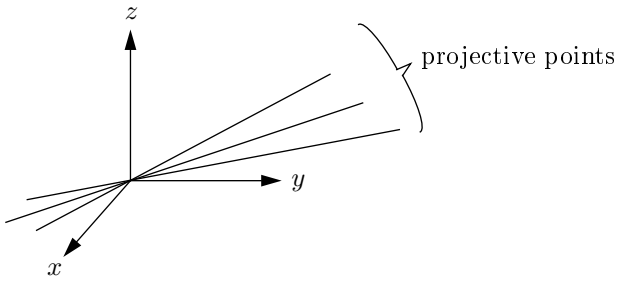


Figure 4.1. Projective points as lines in R^3.

Analogously, we can identify the points in the *real projective plane* P_2 with vectors $(v_1 : v_2 : v_3) \in \mathrm{R}^3$. However, we will consider all vectors $(\lambda v_1 : \lambda v_2 : \lambda v_3)$, $\lambda \in \mathrm{R}$, to represent the same projective point. In other words, we define a projective point as a *one-dimensional linear subspace* of R^3. We will often say "the point $(v_1 : v_2 : v_3)$" when we mean the projective point that has been identified with the subspace spanned by $(v_1 : v_2 : v_3)$. The definition is illustrated in Figure 4.1; the projective points are the lines in R^3 that go through the origin. Note that $(0 : 0 : 0)$ does not span a one-dimensional subspace and consequently, does not represent a projective point.

In P_2, projective lines can be defined in the same way as points, i.e. as one-dimensional subspaces[1] of R^3. Thus, a vector $(v_1 : v_2 : v_3)$ may represent both a projective point and a projective line. However, we will treat point and line as two separate entities and talk about "the point $(v_1 : v_2 : v_3)$" in contrast to "the line $(v_1 : v_2 : v_3)$".

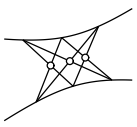
With this concrete definition of projective points and lines, projective geometry can be based on the theory of real numbers and vector spaces. (However, this is certainly not the only possible approach.)

In equations involving points, lines and other geometrical entities, we will treat points and lines as column vectors and write

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

or, in running text, $v = (v_1, v_2, v_3)^T$.

[1] The lines could also be defined as *two*-dimensional subspaces. However, since each two-dimensional subspace of R^3 has a unique one-dimensional orthogonal subspace, the definitions are equivalent.



A point p is *on* or *incident with* a line l if

$$p^T l = p_1 l_1 + p_2 l_2 + p_3 l_3 = 0$$

First we must show that there is exactly one line on two given distinct points p and q, namely

$$l = p \times q$$

From the definition of the cross-product, it is easy to verify that $l = p \times q$ implies $p^T l = 0, q^T l = 0$. To see that l is the *only* line that contains p and q, consider the 2×3 matrix $A = (p, q)^T$. Since any line m that contains both p and q must satisfy $Am = 0$, m must be in the nullspace of A, $N(A)$. The rank of a matrix plus the dimension of its nullspace equals the number of columns. Since p and q represent distinct points they are linearly independent and $\operatorname{rank} A = 2$. Therefore, $\dim N(A) = 1$ which means that $N(A)$ corresponds to a single projective line.

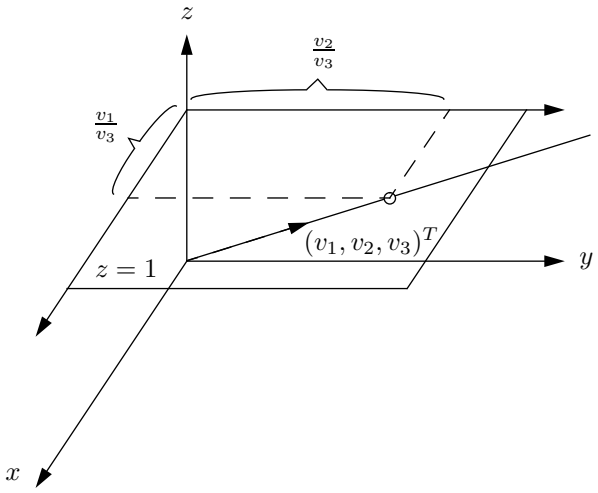


Figure 4.2. The correspondence between projective points and Euclidean points.

Three points are said to be *collinear* if they are on the same line. If p, q and r are distinct they are collinear exactly when p is on the line $q \times r$, i.e., when $p^T(q \times r) = 0$ or equivalently,

$$\begin{vmatrix} p & q & r \end{vmatrix} = 0$$

The set of points on a line is called a *range of points*. Similarly, three lines k, l, m are said to be *concurrent* if they have a point in common. Again, that happens exactly when

$$\begin{vmatrix} k & l & m \end{vmatrix} = 0$$

The set of lines on a point is called a *pencil of lines*.

From the definitions we made earlier, it is clear that a Euclidean point is very different from a projective point. The former is a vector in R^2, the latter a one-dimensional subspace of R^3. However, we can associate a projective point with each Euclidean point:

$$(x, y) \mapsto \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

That allows us to treat the Euclidean points as a subset of the projective points. Conversely,

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \mapsto (\frac{v_1}{v_3}, \frac{v_2}{v_3})$$

if $v_3 \neq 0$. Figure 4.2 shows this relationship geometrically. In R^3, the lines through the origin represent the projective points and the plane $z = 1$ is identified with the Euclidean plane. An R^3 line with direction $(v_1, v_2, v_3)^T$ intersects the plane $z = 1$ in $(v_1/v_3, v_2/v_3)$. This is called the *standard embedding* of the Euclidean plane. It is the simplest but of course not the only way of identifying Euclidean points with projective ones.

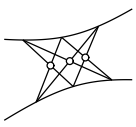
With the standard embedding of the Euclidean plane, a projective point $(v_1, v_2, 0)^T$ corresponds to an R^3 line in the plane $z = 0$. Since that line does not intersect the plane $z = 1$, the projective point has no Euclidean counterpart. Because the Euclidean point corresponding to $(v_1, v_2, \epsilon)^T$ moves further and further away from the Euclidean origin as $\epsilon \to 0$, $(v_1, v_2, 0)^T$ is called a *point at infinity*. However, in the projective plane, $(v_1, v_2, 0)^T$ exists and is no different from any other point. It is an infinity point only with respect to the standard embedding of the Euclidean plane.

A similar relationship can be established between projective and Euclidean lines. A Euclidean line is defined by the equation $ax + by + c = 0$ where a and b are not both zero. The equation can be written

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = 0$$

If we interpret this as an incidence in P_2, it says that the point $(x, y, 1)^T$ is on the line $(a, b, c)^T$. Thus, $(a, b, c)^T$ is the projective counterpart of the given Euclidean line. Figure 4.3 shows the geometric interpretation of this relationship in R^3. π is a plane through the R^3 origin which cuts out a Euclidean line in the plane $z = 1$. If (x, y) is a point on the Euclidean line, the R^3 vector $(x, y, 1)^T$ is in π. Since $(x, y, 1)(a, b, c)^T = 0$, $(a, b, c)^T$ must be the normal of π in R^3.

From Figure 4.3 we see that there is only one plane π that does not cut $z = 1$, namely the xy-plane whose normal is $(0, 0, 1)^T$. Thus, $(0, 0, 1)^T$ is the



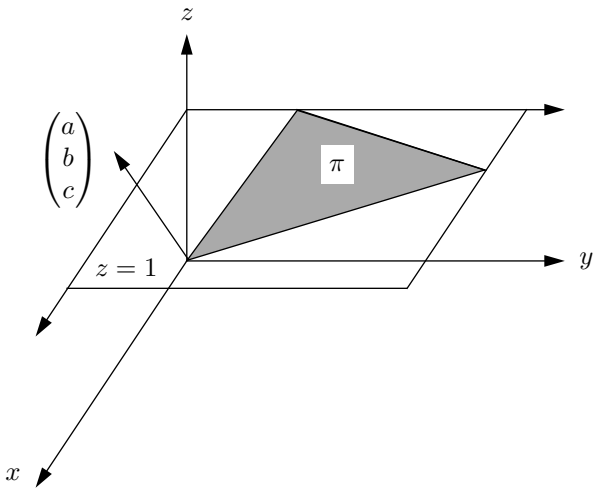


Figure 4.3. The correspondence between projective lines and Euclidean lines.

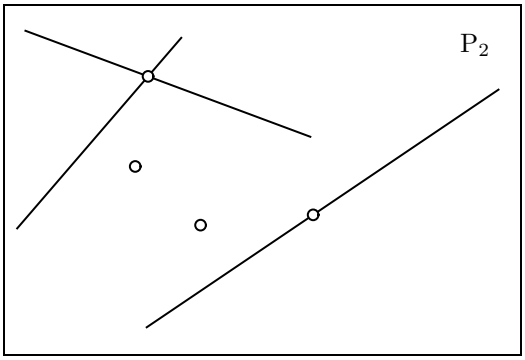


Figure 4.4. Points and lines in the projective plane.

only projective line that does not have a corresponding Euclidean line. Because it contains every point at infinity, it is called the *line at infinity*.

From now on, projective points and lines will only occasionally be depicted as R^3 lines and planes; we will almost always draw them as Euclidean points and lines, as in Figure 4.4.

4.2 Duality

Both projective points and projective lines are represented by vectors in R^3. When the vectors are used in equations, only the context tells us which ones represent points and which ones represent lines. For example, the equation $p^T l = 0$ can be interpreted as "the point p is on the line l" or "the point l is on the line p".

In the previous section, we proved that there is exactly one line that contains any given pair of points. In that proof we could have replaced every word "point" with "line" and every "line" with "point" and all equations would still have been valid. The proof therefore shows that there is exactly one point on two given lines, or in other words, two lines have exactly one point in common.

In general, for every valid statement about points and lines there is a valid *dual* statement where the words "point" and "line" have been interchanged. This relationship between point and lines (hyperplanes in higher dimensions) is characteristic of projective geometry and is referred to as *the principle of duality* or Poncelét-Gergonneduality.

4.3 Projective transformations

A linear, 1-1 transformation $T : \mathrm{P}_2 \to \mathrm{P}_2$ which maps triples considered as points to triples considered as points is called a *projective transformation* or a *projectivity*. T can be represented by a 3×3 matrix M:

$$q = Mp$$

where p and q are vectors representing points. Since T is 1-1, M is non-singular. Furthermore, M and λM, $\lambda \in \mathrm{R}$, represent the same projective transformation since Mp and λMp represent the same projective point.

If Section 4.1 we saw that three points p, q and r are collinear if and only if

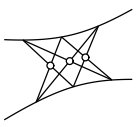
$$\begin{vmatrix} p & q & r \end{vmatrix} = 0$$

Since

$$\begin{vmatrix} Mp & Mq & Mr \end{vmatrix} = \begin{vmatrix} M \begin{pmatrix} p & q & r \end{pmatrix} \end{vmatrix} = |M| \cdot \begin{vmatrix} p & q & r \end{vmatrix}$$

T preserves collinearity. Projective transformations are therefore also called *collineations*.

It should be noted that M can only be applied to triples representing points. In order to preserve incidences between points and lines, triples representing lines



must be transformed with M^{-T} (where $^{-T}$ means invert and transpose). Then, if p is a point on a line l

$$(Mp)^T(M^{-T}l) = p^T M^T M^{-T} l = p^T l = 0$$

Suppose we want to determine a transformation M such that it maps a projective point p_i to another point q_i, i.e.,

$$Mp_i = \lambda_i q_i,\ \lambda_i \in \mathrm{R} \tag{4.1}$$

How many pairs of corresponding points p_i, q_i does it take to completely determine M? Since (4.1) gives us three scalar equations for each i, and since M contains nine unknown elements, it may look as if three pairs would suffice. However, each equation introduces a new unknown λ_i. Therefore, four pair of points are required. (The constant in the last equation, λ_4, can be set to 1 since M is only determined up to a scale factor anyway.) Of course, the equations must be linearly independent, i.e., no three points p_i must be collinear. How can M be computed? Assume first that $p_1 = e_1$, $p_2 = e_2$, $p_3 = e_3$ and $p_4 = u$, where

$$e_1 = \begin{pmatrix}1\\0\\0\end{pmatrix},\ e_2 = \begin{pmatrix}0\\1\\0\end{pmatrix},\ e_3 = \begin{pmatrix}0\\0\\1\end{pmatrix},\ u = \begin{pmatrix}1\\1\\1\end{pmatrix} \tag{4.2}$$

Then Equation 4.1 for $i = 1, 2, 3$ can be written

$$M\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix} = \begin{pmatrix}\lambda_1 q_1 & \lambda_2 q_2 & \lambda_3 q_3\end{pmatrix}$$

When combining that with Equation 4.1 for $i = 4$ and $\lambda_4 = 1$ we get

$$M\begin{pmatrix}1\\1\\1\end{pmatrix} = \lambda_1 q_1 + \lambda_2 q_2 + \lambda_3 q_3 = \begin{pmatrix}q_1 & q_2 & q_3\end{pmatrix}\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix} = q_4 \tag{4.3}$$

Thus, we can first calculate

$$\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix} = \begin{pmatrix}q_1 & q_2 & q_3\end{pmatrix}^{-1} q_4$$

and then

$$M = \begin{pmatrix}\lambda_1 q_1 & \lambda_2 q_2 & \lambda_3 q_3\end{pmatrix}$$

(Since q_1, q_2, q_3 are not collinear, the matrix (q_1, q_2, q_3) has full rank and is therefore invertible.) Now, if p_i, $i = 1, 2, 3, 4$ are arbitrary points we compute the projectivity M_1 which maps e_1, e_2, e_3, u to p_1, p_2, p_3, p_4 and the projectivity M_2 which maps e_1, e_2, e_3, u to q_1, q_2, q_3, q_4. The required map is then $M = M_2 M_1^{-1}$.

By the principle of duality, the projectivity M can also be computed from four pairs of corresponding lines, no three concurrent. It can also be computed from three pairs of corresponding points and one pair of corresponding lines, which is more natural for example in affine geometry where one line is fixed, see Section 4.12.4. (In Equation 4.3, q_1, q_2, q_3 would have to be replaced by the columns of $(q_1, q_2, q_3)^{-T}$ and λ_i by $1/\lambda_i$.)

The fact that a projectivity on P_2 is completely determined by four pairs of corresponding points is also known as the *fundamental theorem of projective geometry*.

4.4 Homogeneous coordinates

In Section 4.1, points in P_2 were defined as one-dimensional linear subspaces of R^3, and these subspaces were represented by the R^3 vectors spanning them. Thus, we could say that the R^3 vectors *are* the projective points. In this section, we will introduce projective coordinate systems or *projective frames* for P_1 and P_2. A projective frame is a geometric object that can be used as a reference in order to assign unique numerical coordinates to every projective point. These coordinates, often called the *homogeneous coordinates*, will also be vectors in R^3. However, in contrast to the projective points themselves, the coordinate vectors are meaningful only in combination with a projective frame.

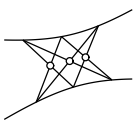
We saw in the previous section that four pairs of corresponding points determine a projectivity. Therefore, we define a projective frame to be an ordered list of four points $F = p_1, p_2, p_3, p_4$, no three points collinear. The *standard frame* is $F_0 = e_1, e_2, e_3, u$ where e_1, e_2, e_3 and u were defined in the previous section (Equation 4.2). Let T be the unique projectivity that maps F to F_0. The homogeneous coordinates of a point p relative to the frame F, denoted $[p]_F$, is then the vector representing Tp. Since Tp has many representatives, $[p]_F$ is only determined up to a scalar factor. The point p_4, which is mapped to u by T, is called the *unit point* of the frame F.

If we consider the vectors in F and F_0 as lines, we can use the same definition for line coordinates.

4.5 Correlations and polars

A projectivity, or collineation, was defined above as a linear, 1-1 linear transformation mapping triples considered as points to triples considered as points. A similar mapping from triples considered as points to triples considered as lines is called a *correlation*. Just like a collineation, a correlation can be represented by a non-singular matrix M.

Correlations which satisfy the following constraint are called *polarities*: if p and q are two points, p will be on the line Mq if and only if q is on the line Mp,



i.e.,

$$\forall\, p, q \in \mathrm{P}_2 \colon q^T M p = 0 \Leftrightarrow p^T M q = 0$$

Since $q^T M p$ is a scalar, the left-hand side of the equivalence can be written $(q^T M p)^T = p^T M^T q = 0$. Since the equivalence must hold for all points p and q, $M = M^T$. Thus, polarities are represented by symmetric matrices. The line $l = Mp$ is called the *polar* of p, and conversely, p is called the *pole* of l, with respect to the polarity M.

4.6 Perspectivities

A projectivity $T \colon \mathrm{P}_2 \to \mathrm{P}_2$ is a *perspectivity* if

$$\exists\, q \in \mathrm{P}_2, \forall\, p \in \mathrm{P}_2 \colon \begin{vmatrix} q & p & Tp \end{vmatrix} = 0$$

If q is a point, it is called the *center of perspectivity*. The definition then says that any point p and its image must be on a line through the center of perspectivity, see Figure 4.5a. Dually, if we interpret q as a line, the line p and its image must be concurrent with q, see Figure 4.5b.

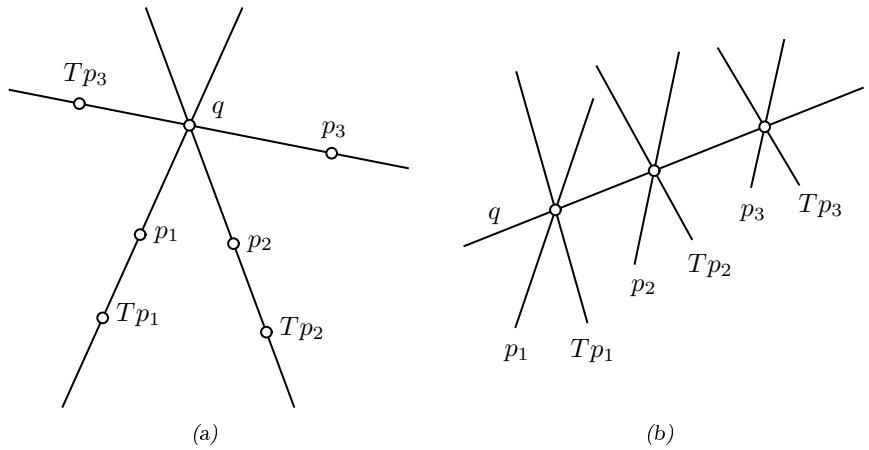


Figure 4.5. Perspectivities in the plane.

A projectivity T which leaves three concurrent lines (or three collinear points) invariant is a perspectivity. To prove that, pick three points p_1, p_2, p_3, one on each line. They must be different from q and not collinear (Figure 4.5a). Then, if

$$p_3 = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 q,\ \ \lambda_i \in \mathrm{R}$$

we get

$$Tp_3 = \lambda_1 T p_1 + \lambda_2 T p_2 + \lambda_3 q$$

Since q, p_3 and Tp_3 are collinear

$$\begin{aligned}\left|q \quad p_3 \quad Tp_3\right| &= \left|q \quad (\lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 q) \quad (\lambda_1 Tp_1 + \lambda_2 Tp_2 + \lambda_3 q)\right| \\ &= \left|q \quad \lambda_1 p_1 \quad \lambda_2 Tp_2\right| + \left|q \quad \lambda_2 p_2 \quad \lambda_1 Tp_1\right| \\ &= \lambda_1\lambda_2(\left|q \quad p_1 \quad Tp_2\right| + \left|q \quad p_2 \quad Tp_1\right|) = 0\end{aligned}$$

Since p_3 is not on the same line as p_1 or p_2, $\lambda_1 \neq 0$, $\lambda_2 \neq 0$. Hence, $|q\ p_1\ Tp_2| + |q\ p_2\ Tp_1| = 0$. Similarly, for an arbitrary point p in the plane we get

$$\left|q \quad p \quad Tp\right| = const \cdot (\left|q \quad p_1 \quad Tp_2\right| + \left|q \quad p_2 \quad Tp_1\right|)$$

hence

$$\left|q \quad p \quad Tp\right| = 0$$

Conversely, a perspectivity leaves every line on a certain point (here, q) invariant. That follows immediately from the definition of perspectivity.

The perspectivities do not form a subgroup of the full projective group. In fact, it can be shown that any projectivity can be written as a finite sequence of perspectivities.

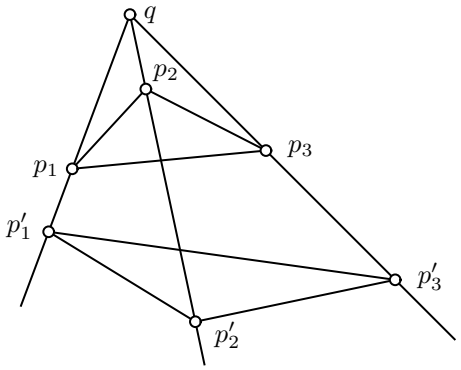
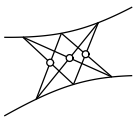


Figure 4.6. Perspective triangles.

If a plane figure can be mapped onto another by a perspectivity, the two figures are said to be *perspective*. For example, the triangles $p_1p_2p_3$ and $p'_1p'_2p'_3$ in Figure 4.6 are perspective. (The projectivity which maps q, p_1, p_2, p_3 to q, p'_1, p'_2, p'_3 leaves the lines qp_1, qp_2 and qp_3 invariant and is therefore a perspectivity.)

Interestingly, the two triangles in Figure 4.6 are also perspective from a line because the intersections of corresponding sides are collinear (Figure 4.7). This is known as the theorem of Desargues. We will just outline a proof here. Let p_1, p_2, p_3, q be the standard frame F_0. We can then write down the coordinates of l_1, l_2, l_3 and the coordinates of the sides of the triangle $p_1p_2p_3$. Since p'_i is on



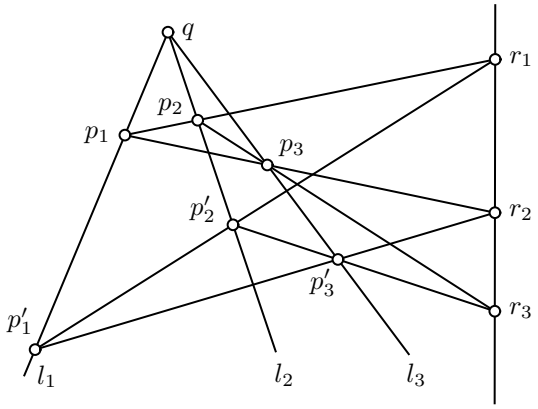


Figure 4.7. The theorem of Desargues.

l_i, $l_i^T p_i' = 0$. It follows that

$$p_1' = \begin{pmatrix} p_{11}' \\ p_{12}' \\ p_{12}' \end{pmatrix}, p_2' = \begin{pmatrix} p_{21}' \\ p_{22}' \\ p_{21}' \end{pmatrix}, p_3' = \begin{pmatrix} p_{31}' \\ p_{31}' \\ p_{33}' \end{pmatrix}$$

From this, the coordinates of the sides of $p_1'p_2'p_3'$ and the intersection points r_1, r_2, r_3 are easily calculated. Finally, we verify that $|r_1\ r_2\ r_3| = 0$, which proves that r_1, r_2, r_3 are collinear.

4.7 The projective line

We will be concerned primarily with the projective plane, but occasionally we will also have to work with the one-dimensional projective space P_1, often called the *projective line*. In analogy with P_2, we define the points in P_1 as one-dimensional subspaces of R^2. When we talk about the point $p = (p_1, p_2)^T$, we mean the projective point that has been identified with the subspace of R^2 that is spanned by $(p_1, p_2)^T$.

A P_1 projectivity is a linear 1-1 mapping $T : \mathrm{P}_1 \to \mathrm{P}_1$. It can be represented by a 2×2 non-singular matrix. We can use the same method as in Section 4.3 to compute a projectivity from pairs of corresponding points. However, in P_1, a projectivity is completely determined by *three* pairs.

Coordinates in P_1 are defined in the same way as in P_2. A frame F in P_1 consists of three distinct points p_1, p_2, p_3, where p_3 is called the *unit point*. The standard frame F_0 consists of the points $(1,0)^T$, $(0,1)^T$ and $(1,1)^T$. If T is the projectivity that maps F to F_0, the homogeneous coordinates of a point p is $[p]_F = Tp$.

The points in P_1 are often identified with points or lines in P_2. Actually, when we draw P_1 as a line on a paper (Figure 4.8a), we have mapped the elements of

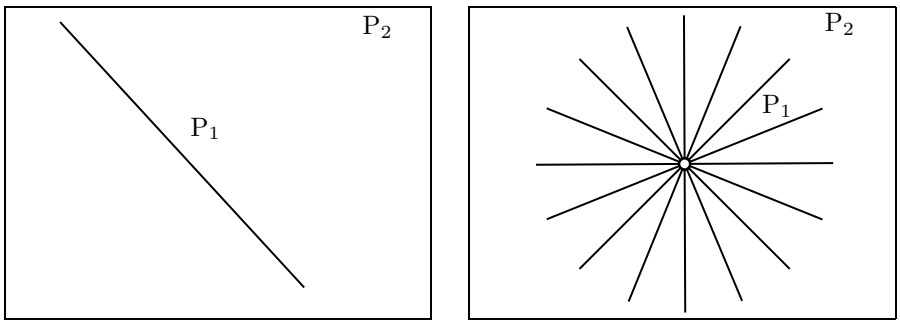


Figure 4.8. P_1 *can be embedded in* P_2 *either as a range of points (a) or as a pencil of lines (b).*

P_1 onto collinear points in the plane. Algebraically, this amounts to multiplying each 2-vector p representing a point in P_1 by a 3×2 matrix A of rank 2:

$$q = Ap \tag{4.4}$$

Since $\operatorname{rank} A = 2$, the three vectors Ap_1, Ap_2 and Ap_3 will always be linearly dependent, and thus represent collinear points in P_2. Between P_1 and the points on the P_2 line, the mapping A is 1-1. We say that P_1 is *embedded* as a *range of points* in P_2 by A.

By the principle of duality, we may also interpret Ap as a *line*. We then say that P_1 has been embedded as a pencil of lines (Figure 4.8b).

Let a_1, a_2, a_1' and a_2' be four distinct points on a line l in P_2. Then both $A = (a_1, a_2)$ and $A' = (a_1', a_2')$ will map points in P_1 to points on l. Since a_1, a_2, a_1', a_2' are collinear they are linearly dependent and we can write

$$a_1' = m_{11}a_1 + m_{21}a_2$$
$$a_2' = m_{12}a_1 + m_{22}a_2$$

or $A' = AM$ where

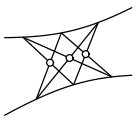
$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$$

Because a_1' and a_2' are distinct, $|M| \neq 0$. Hence, M can be considered as a projectivity on P_1. If $u \in P_1$ and $v = Mu$, we get

$$A'u = AMu = Av$$

Thus, a change of embedding corresponds to a projectivity on P_1.

Suppose a projectivity $T : P_2 \to P_2$ maps a line l onto another line m. Then the correspondence between points on l and points on m can be described by



a projectivity on P_1. To be more specific, if A embeds P_1 as the line l and B embeds P_1 as the line m,

$$\exists\, M\colon \mathrm{P}_1 \to \mathrm{P}_1, \forall\, u \in \mathrm{P}_1\colon TAu = BMu$$

To see that, choose two distinct points p_1 and p_2 on l. Let $A = (p_1, p_2)$ and $B = (Tp_1, Tp_2)$. Then $B = T(p_1, p_2) = TA$ and $\forall u \in \mathrm{P}_1\colon TAu = Bu$. The choice of A was arbitrary, but the choice of B was not. However, if B' is an arbitrary embedding of m we saw above that $B = B'M$, for some $M \in \mathrm{R}^{2\times 2}, |M| \neq 0$. For this M, $TAu = B'Mu$ for every u. Since M is non-singular it is a projectivity on P_1. That completes the proof. Of course, the same applies to correlations on P_2. Then T maps points to lines and B embeds P_1 as a pencil of lines.

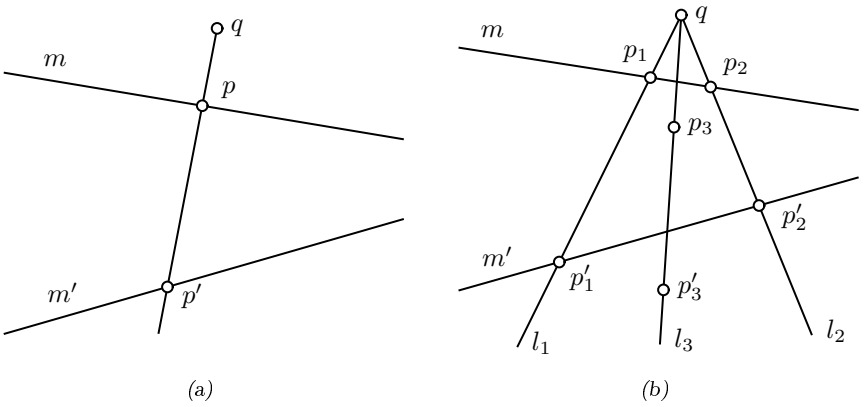


Figure 4.9. Projecting points on a line m onto points on a line m'.

Figure 4.9a shows a common geometric construction. Let m and m' be two fixed lines and q a fixed point. For each point p on m, draw the line from q to p. This line intersects m' in p'. How are p and p' related algebraically? Choose three distinct lines l_1, l_2, l_3 on q. The line l_1 intersects m and m' in p_1 and p'_1, see Figure 4.9b. The line l_2 intersects m and m' in p_2 and p'_2. Let p_3 and p'_3 be two distinct points on l_3 not on m or m'. By the fundamental theorem (Section 4.3) there is a unique projectivity $T : \mathrm{P}_2 \to \mathrm{P}_2$ which maps $qp_1p_2p_3$ to $qp'_1p'_2p'_3$. Obviously, T maps m onto m' and l_1, l_2, l_3 onto themselves. Since it leaves three lines on q invariant it is a perspectivity, with q as the center. Consequently, it leaves *all* lines on q invariant. Thus, T represents the geometric construction in Figure 4.9a and $p' = Tp$ in P_2 coordinates. We showed above that the correspondence between points on m and m' defined by a P_2 projectivity T can be represented by a projectivity $M\colon \mathrm{P}_1 \to \mathrm{P}_1$. Thus, in P_1 coordinates, $p = Mp'$. (The perspectivity T is not completely determined by m, m' and q. However, every perspectivity which maps m to m' and which has q as its center induces the same projectivity M on P_1.)

We can now show that any projectivity $M\colon \mathrm{P}_1 \to \mathrm{P}_1$ can be represented by no more than three successive perspectivities in P_2. Suppose that A embeds P_1

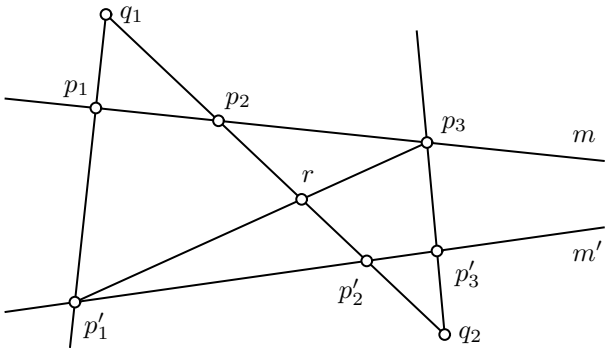


Figure 4.10. Two successive perspectivities project any three points on a line m onto any three points on another line m', if the lines m and m' are distinct.

as the line m and A' embeds P_1 as m', see Figure 4.10. Choose three distinct points points $u_1, u_2, u_3 \in \mathrm{P}_1$. Let $p_i = Au_i$ and $p'_i = A'Mu_i$. We now construct a projectivity in P_2 which maps p_i to p'_i. Let q_1 be the intersection of $p_1p'_1$ and $p_2p'_2$ and let q_2 be the intersection $p_2p'_2$ and $p_3p'_3$. p_1, p_2, p_3 can be projected from q_1 to p'_1, r, p_3 on the line p'_1p_3. Those points can in turn be projected from q_2 onto p'_1, p'_2, p'_3. If m and m' are identical this construction does not work. In that case, we must first project p_1, p_2, p_3 to a different line. We just saw above that these projections correspond to perspectivities in P_2. Thus, there is a P_2 projectivity T consisting of no more than three successive perspectivities which maps p_1, p_2, p_3 onto p'_1, p'_2, p'_3. Since a projectivity on P_1 is completely determined by three pairs of corresponding points, the P_1 projectivity induced by T must be M.

4.8 Conics

4.8.1 Definition

A conic (ellipse, hyperbola or parabola) in the Euclidean plane is defined by the second-order polynomial

$$ax^2 + by^2 + 2cxy + 2dx + 2ey + f = 0 \tag{4.5}$$

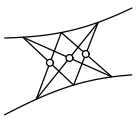
In matrix form, this equation can be written

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} a & c & d \\ c & b & e \\ d & e & f \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = 0 \tag{4.6}$$

or

$$p^T C p = 0 \tag{4.7}$$

Note that C is symmetric, i.e., $C^T = C$.



Multiplying p or C by a scalar does not affect Equation 4.7. We can therefore interpret the equation as a constraint on a projective point p. Analogous to the definition of a projective point, we define a *projective conic* to be a *one-dimensional subspace of* $\mathrm{R}^{3\times3}$ *spanned by a symmetric matrix*. If the matrix is non-singular, the conic is *proper*, otherwise *degenerated* (Section 4.8.13). A projective point that satisfies (4.7) is said to be *on* the conic C.

4.8.2 The real and imaginary unit circles

The matrix

$$C_u = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

corresponds to the equation $x^2 + y^2 - 1 = 0$, the Euclidean unit circle. The identity matrix,

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

corresponds to the equation $x^2 + y^2 + 1 = 0$, sometimes called the *imaginary unit circle*, which contains no real points.

4.8.3 Tangents and intersecting lines

A line l and a conic C *intersect* in a point p if p is on both l and C, i.e., if

$$\begin{cases} p^T l = 0 \\ p^T C p = 0 \end{cases} \tag{4.8}$$

If q_1 and q_2 are two distinct points on l, every point on l (except q_2) can be written

$$p = q_1 + \lambda q_2$$

where λ is a scalar. If p is on the conic, then

$$p^T C p = q_2^T C q_2 \lambda^2 + 2 q_2^T C q_1 \lambda + q_1^T C q_1 = 0 \tag{4.9}$$

Provided that q_2 is not on the conic, $q_2^T C q_2 \neq 0$. (4.9) is then a second-order equation in λ. Consequently, (4.8) has two roots which means that C and l have two points in common (not necessarily real or distinct), see Figure 4.11.

If C is proper and p is a double root of (4.8), we say that the line l is a (proper) *tangent* to C in p. That happens if and only if

$$(q_2^T C q_1)^2 = (q_1^T C q_1)(q_2^T C q_2) \tag{4.10}$$

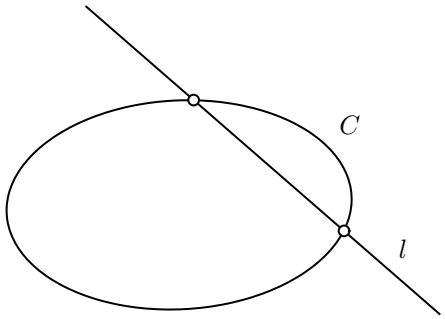


Figure 4.11. A line intersects a conic in two points.

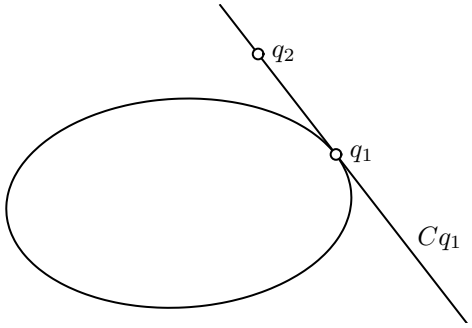


Figure 4.12. A tangent.

Now, let q_1 be a fixed point on C. Then $q_1^T C q_1 = 0$ and Cq_1 can be interpreted as a line through q_1. From (4.10) we see that the line trough q_1 and q_2 is tangent to C if and only if $q_2^T C q_1 = 0$, i.e., if and only if q_2 is on the line Cq_1. In other words, if C is proper, the line Cq_1 is tangent to C and it is the *only* tangent through q_1, see Figure 4.12. (This is not true for degenerated conics.)

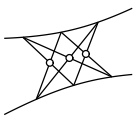
4.8.4 Line conics

So far we have considered p in Equation 4.7 as a point. When we make that interpretation, we will call C a *point conic*. It consists of the points $\{p\colon p^T C p = 0\}$. However, we can equally well interpret (4.7) as a constraint on a projective *line* p. In that case, C is a *line conic* which consists of the lines $\{l\colon l^T C l = 0\}$.

From the dual of the argument above it is clear that for every point q, a line conic contains two lines (not necessarily real or distinct) that intersect in q, see Figure 4.13. If C is proper and the system

$$\begin{cases} p^T l = 0 \\ l^T C l = 0 \end{cases} \tag{4.11}$$

has a double root l, the corresponding point p is called an *envelope point* of the line conic C. Furthermore, Cl is the *only* envelope point on the line l.



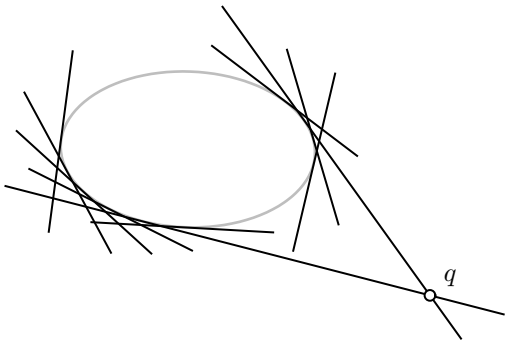


Figure 4.13. A line conic has two lines which intersect in any given point q.

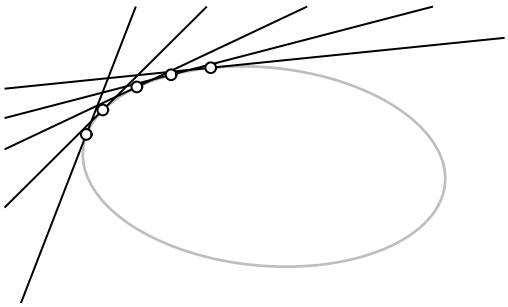


Figure 4.14. If C is proper, the line conic C^{-1} consists of the tangents of the point conic C.

There is a 1-1 correspondence between proper point conics and proper line conics. Let p be a point on a point conic C, $|C| \neq 0$, and $l = Cp$ the tangent through p. From

$$p^T Cp = p^T CC^{-1}Cp = (C^T p)^T C^{-1}Cp = (Cp)^T C^{-1}Cp = l^T C^{-1} l = 0$$

we see that the line conic C^{-1} consists of the tangents of the point conic C. Conversely, the point conic C consists of the envelope points of the line conic C^{-1}, see Figure 4.14. Because of this, we do not always distinguish between proper point conics and line conics; we just say "the conic C" when we mean the point conic C or the line conic C^{-1}. However, no such 1-1 correspondence exists between degenerated conics since C is not invertible if $|C| = 0$. When referring to a degenerated conic we must make it clear whether we are treating it as a point conic or as a line conic.

4.8.5 Conics as polarities

Since a proper conic is represented by a symmetric, non-singular matrix, it can be considered a polarity (Section 4.5). Thus, the polar of a point p with respect to a conic C is the line Cp. Equation 4.7 shows that the points on the conic are

exactly the points that are on their own polars, i.e., the *self-polar* points. From the result of Section 4.8.3 it follows that the polar of a point on the conic is the tangent of the conic in that point.

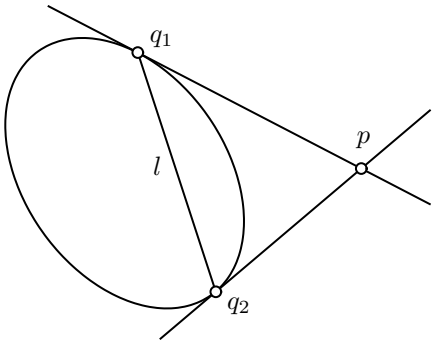


Figure 4.15. Pole and polar.

In Figure 4.15, q_1 and q_2 are two points on a conic C. The tangent through q_1 is also the polar of q_1. Since p is on the polar of q_1, q_1 must be on the polar of p. The same goes for q_2. Thus, the polar of p is the line l between q_1 and q_2.

4.8.6 Interior and exterior points

A conic with real coefficients divides the real projective plane into *interior* and *exterior* points. A point is exterior if its polar intersects the conic in two real points. Otherwise, the point is interior.

4.8.7 Projections of conics

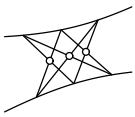
How is a conic affected by a projectivity? If the projectivity is represented by a point transformation matrix M, C is a point conic, p is a point, $q = Mp$, and $C' = M^{-T}CM^{-1}$, we can write

$$p^TCp = p^TM^TM^{-T}CM^{-1}Mp = q^TM^{-T}CM^{-1}q = q^TC'q \tag{4.12}$$

Since C is symmetric, C' is also symmetric and thus represents a conic. C' is degenerated only if C is. From Equation 4.12 we see that q is on C' if and only if p is on C. Thus, C' is the projection of the point conic C.

We have just showed that a projectivity maps conics to conics. But given two conics C and C', is it always possible to find a projectivity that maps C to C'? A real, non-singular, symmetric matrix C can always be diagonalized by an orthogonal matrix U ($U^TU = E$):

$$D = U^TCU \tag{4.13}$$



where D is diagonal (see e.g. [Leon86] for a proof). Furthermore, if the diagonal elements of D are d_1, d_2 and d_3, they can be scaled to unit magnitude with

$$D' = S^T D S \tag{4.14}$$

where

$$S = \begin{pmatrix} \frac{1}{\sqrt{|d_1|}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{|d_2|}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{|d_3|}} \end{pmatrix}$$

The elements of D' can be shifted along the diagonal:

$$D'' = V^T D' V \tag{4.15}$$

where

$$V = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Finally, we can change the sign of all diagonal elements by multiplying D'' by -1. Thus, the projectivity $\pm(USV)^{-1}$ maps the conic C either onto the unit circle,

$$C_u = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

or onto the imaginary unit circle

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

However, E cannot be projected onto C_u by a real transformation. This is a consequence of Sylvester's theorem. What sets C_u and E apart is the difference between the number of minus signs and the number of plus signs on the diagonal elements. That number can only be changed by a complex projectivity, for example

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -i \end{pmatrix}$$

To summarize, if we can find real projectivities which map both C_1 and C_2 onto C_u or both C_1 and C_2 onto E, there is a real projectivity which maps C_1 to C_2.

4.8.8 Conics in the standard embedding

With the standard embedding, a proper conic with real points in the Euclidean plane is cut out by a double cone in R^3, see Figure 4.16. The tip of the cone is always at the R^3 origin, but the cone is not necessarily circular. If the xy-plane does not intersect or touch the cone, the corresponding conic in $z = 1$ will be an ellipse (Figure 4.16a). If the xy-plane intersects the cone, the corresponding conic will be a hyperbola (Figure 4.16b). Since the xy-plane represents the projective line at infinity, the two R^3 lines in which the xy-plane intersects the cone represent two projective points at infinity. Thus, projectively, a hyperbola intersects the line at infinity in two real points. Figure 4.16c shows the limiting case where the cone just touches the xy-plane. The corresponding conic is a parabola. Projectively, the line at infinity is tangent to every parabola.

By diagonalizing the matrix of the conic with an orthogonal matrix (Equation 4.13), we rotate the double cone in R^3 so that its symmetry axis coincides with the z-axis. Scaling the diagonal elements to unit magnitude (Equation 4.14) makes the cone circular. Shifting the elements on the diagonal (Equation 4.15) turns a hyperbola into an ellipse or vice versa. (Actually, V is orthogonal and rotates the double cone 90 degrees in R^3.)

4.8.9 Determining the coefficients of a conic

We will often need to determine the conic that goes through certain points or touches certain lines. What constraints are sufficient to do that? The coefficient matrix C of the conic contains nine elements. Since C is only determined up to a constant factor and since C is symmetric, it contains only five independent elements. Therefore, five points on the conic will be sufficient for computing C. Let

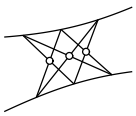
$$p_i = \begin{pmatrix} p_{i1} \\ p_{i2} \\ p_{i3} \end{pmatrix}, \; i = 1, \ldots, 5$$

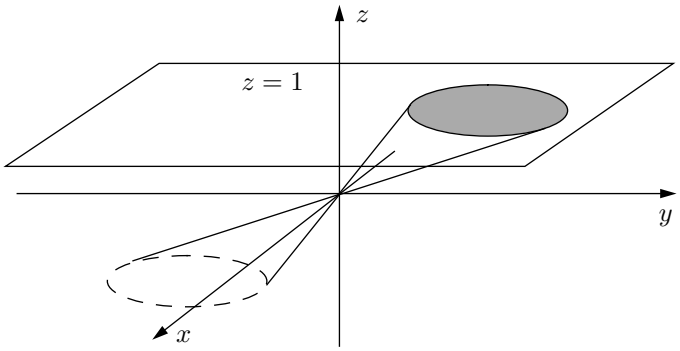
be the coordinates of these five points. Using the homogeneous form of Equation 4.5 we get

$$ap_{i1}^2 + bp_{i2}^2 + 2cp_{i1}p_{i2} + 2dp_{i1}p_{i3} + 2ep_{i2}p_{i3} + fp_{i3}^2 = 0, \; i = 1, \ldots, 5$$

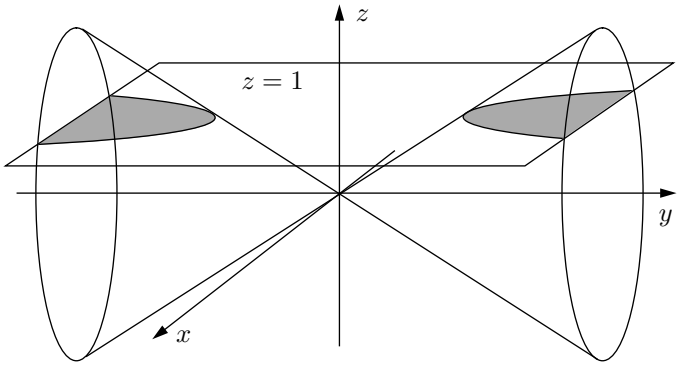
If we define

$$A = \begin{pmatrix} p_{11}^2 & p_{12}^2 & 2p_{11}p_{12} & 2p_{11}p_{13} & 2p_{12}p_{13} & p_{13}^2 \\ p_{21}^2 & p_{22}^2 & 2p_{21}p_{22} & 2p_{21}p_{23} & 2p_{22}p_{23} & p_{23}^2 \\ p_{31}^2 & p_{32}^2 & 2p_{31}p_{32} & 2p_{31}p_{33} & 2p_{32}p_{33} & p_{33}^2 \\ p_{41}^2 & p_{42}^2 & 2p_{41}p_{42} & 2p_{41}p_{43} & 2p_{42}p_{43} & p_{43}^2 \\ p_{51}^2 & p_{52}^2 & 2p_{51}p_{52} & 2p_{51}p_{53} & 2p_{52}p_{53} & p_{53}^2 \end{pmatrix}, \; v = \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

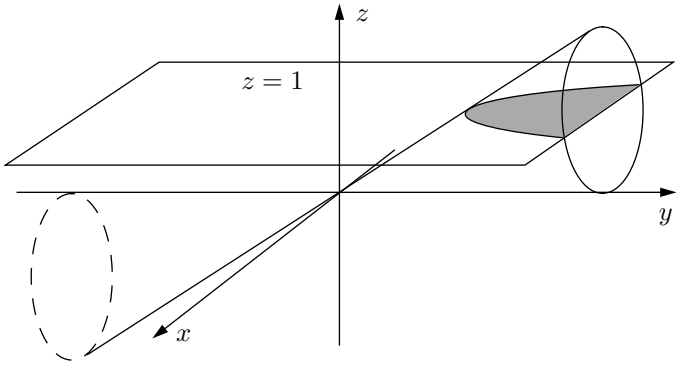




(a) An ellipse.



(b) A hyperbola.



(c) A parabola.

Figure 4.16. Conics in the standard embedding.

these five equations can be written

$$Av = 0 \tag{4.16}$$

Hence, v must be in the nullspace of A. The nullspace of A is the orthogonal complement of the row space of A, i.e., $N(A) = R(A^T)^\perp$. Assume that the five rows of A are linearly independent. (They will be if the points p_i are in general positions.) The row space will then be five-dimensional and since A has six columns, $N(A)$ will be one-dimensional. This means there is only one conic on the five given points, and that this conic is given by a base vector of $N(A)$. A vector $q = (q_1, \ldots, q_6)^T$ is in the rowspace of A if and only if $|B| = 0$ where

$$B = \begin{pmatrix} A \\ q \end{pmatrix}$$

Let A_j, $1 \leq j \leq 6$ be the determinant of the 5×5 submatrix of A resulting from the removal of column j. Since $|B| = q_1 A_1 + \cdots + q_6 A_6$ we see that q is in the row space of A if and only if q is orthogonal to

$$n = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{pmatrix}$$

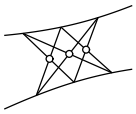
Obviously, n spans $N(A)$, and the coefficients of the conic are given by $v = n$. The resulting coefficient matrix is

$$C = \begin{pmatrix} A_1 & A_3 & A_4 \\ A_3 & A_2 & A_5 \\ A_4 & A_5 & A_6 \end{pmatrix} \tag{4.17}$$

Note that this result holds even if the conic is degenerated because three of the points are collinear. However, if two of the points are identical, $\operatorname{rank} A < 5$ and $\dim N(A) > 1$. That means that there is a pencil of conics on the given points, and Equation 4.17 is not directly applicable.

It is possible to compute C without evaluating all six determinants A_j. However, Equation 4.17 expresses C as a *continuous* function of the given five points. This will prove to be important in Section 5.2.5.

By the principle of duality, five distinct tangent lines are also sufficient for determining a conic. A conic is also determined by three points and the tangents in two of them. Intuitively, a tangency corresponds to a double point. Thus, each tangent gives us an extra point, and with the three given points, we have the necessary five points. For the same reason, four points and the tangent in one of them also determine a conic. However, if the given tangents are not on the given points, there is more than one solution. For example, on four given points there are *two* conics which are tangent to an arbitrary line.



4.8.10 Common points and lines of two conics

Consider the problem of finding the common points of two distinct conics C_1 and C_2. In Figure 4.17, there are four real intersection points. Actually, two conics always intersect in four points, although the points are not necessarily real and distinct. For example, two concentric circles intersect in four complex points.

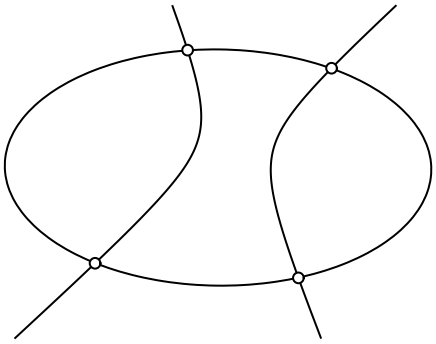


Figure 4.17. Two conics have four points in common.

In order to find the points of intersection, we have to solve the following system of quadratic equations

$$\begin{cases} p^T C_1 p = 0 \\ p^T C_2 p = 0 \end{cases}$$

where C_1 and C_2 are real and symmetric. Our strategy is to diagonalize both matrices simultaneously, thereby transforming the system into a particularly simple form.

First, we project C_1 onto the imaginary unit circle E.

$$V^T C_1 V = E$$

As explained in Section 4.8.7, V will in general be complex. By substituting $p = Vq$ we obtain

$$\begin{cases} p^T C_1 p = q^T V^T C_1 V q = q^T q = 0 \\ p^T C_2 p = q^T V^T C_2 V q \end{cases}$$

The matrix $A = V^T C_2 V$ is complex and satisfies $A^T = A$. It can also be diagonalized: $W^T A W = D$, where D is diagonal and $W^T W = E$. With a new substitution $q = Wr$, we get

$$\begin{cases} q^T q = r^T W^T W r = r^T r = 0 \\ q^T A q = r^T W^T A W r = r^T D r = 0 \end{cases}$$

If $r = (x_1, x_2, x_3)^T$ and d_i the diagonal element of D, we get

$$\begin{cases} r^T r = x_1^2 + x_2^2 + x_3^2 = 0 \\ r^T D r = d_1 x_1^2 + d_2 x_2^2 + d_3 x_3^2 = 0 \end{cases}$$

This is a system of linear equations in x_i^2, whose solution is

$$\begin{pmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \end{pmatrix} = \begin{pmatrix} d_2 - d_3 \\ d_3 - d_1 \\ d_1 - d_2 \end{pmatrix}$$

Hence,

$$\begin{cases} x_1 = \pm\sqrt{d_2 - d_3} \\ x_2 = \pm\sqrt{d_3 - d_1} \\ x_3 = \pm\sqrt{d_1 - d_2} \end{cases}$$

There are eight different sign combinations, which correspond to eight roots $r_1, \ldots, r_8$. However, with suitable indices, we have $r_1 = -r_5$, $r_2 = -r_6$, $r_3 = -r_7$, and $r_4 = -r_8$. Since the roots represent projective points, there are in fact only four roots, $r_1, \ldots, r_4$.

4.8.11 Linear combinations of conics

In the previous section we saw that two conics C_1 and C_2 intersect in four points p_1, p_2, p_3, p_4, not necessarily real or distinct. Any linear combination of C_1 and C_2 also contains these points since

$$p_i^T(\lambda_1 C_1 + \lambda_2 C_2)p_i = \lambda_1 p_i^T C_1 p_i + \lambda_2 p_i^T C_2 p_i = 0$$

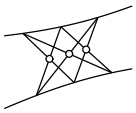
where λ_1 and λ_2 are scalar values.

It is also easy to see that any conic which contains p_1, p_2, p_3, p_4 can be written $\lambda_1 C_1 + \lambda_2 C_2$ for some λ_1, λ_2. From p_1, p_2, p_3, p_4, compute the matrix A in Equation 4.16. In this case, A will only have four rows. The row space of A will be four-dimensional, hence $\dim N(A) = 2$. Since C_1 and C_2 are distinct, the corresponding vectors v in Equation 4.16 are linearly independent and therefore they span $N(A)$.

The set $\{\lambda_1 C_1 + \lambda_2 C_2 \colon \lambda_1, \lambda_2 \in \mathrm{R}\}$ is called a *pencil of conics*.

4.8.12 Steiner's theorem

Consider Figure 4.18a. p and q are two arbitrary points on a proper conic C. Let l be an arbitrary line through p, and r the other intersection point of l and C. Let m be the line through r and q. We have then associated every line l in the pencil of lines on p with a line m in the pencil of lines on q. *Steiner's theorem* states that the two pencils are related by a projectivity on P_1. Let us prove that. Let s be



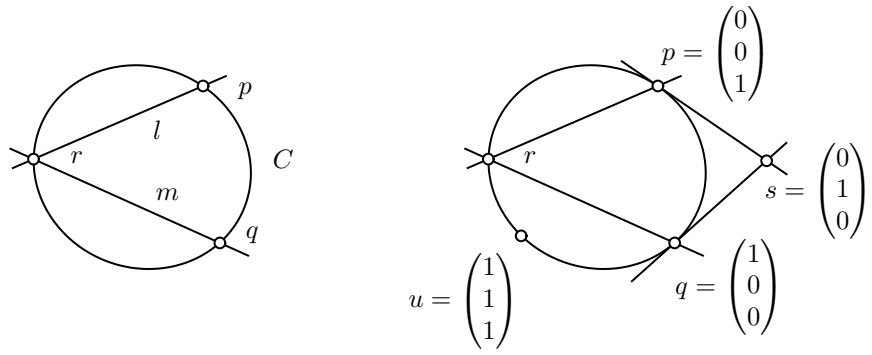


Figure 4.18. Steiner's theorem.

the intersection of the tangents through p and q, and let u be a third, arbitrary point on C, see Figure 4.18b. Choose a frame in P_2 such that $q = (1,0,0)^T$, $s = (0,1,0)^T$, $p = (0,0,1)^T$, and $u = (1,1,1)^T$. In that frame

$$C = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -2 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

(It is easy to verify that p, q, u is on C and that the tangents of C in p and q intersect in s. Since a conic is completely determined by three of its points and the tangents in two of them (Section 4.8.9), C must represent the conic in Figure 4.18b.) Furthermore, l and m must have the form $l = (l_1, l_2, 0)^T, m = (0, m_2, m_3)^T$. Thus, we can consider P_1 as a pencil of lines on p embedded (Section 4.7) by

$$\begin{pmatrix} l_1 \\ l_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}$$

but also a pencil of lines on q embedded by

$$\begin{pmatrix} 0 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} m_2 \\ m_3 \end{pmatrix}$$

If $(m_2, m_3)^T = (l_1, l_2)^T$, it is readily verified that $l \times m$ is on C, i.e., $l \times m = r$. Thus, for this particular choice of embedding, the two line pencils are related by the identity projectivity on P_1. With another embedding, the two pencils would be related by a general projectivity on P_1 (cf Section 4.7).

A consequence of Steiner's theorem is that a projectivity on P_2 which leaves a proper conic invariant and maps three points on the conic onto themselves is the identity projection E. To see that, consider Figure 4.19. p_1, p_2, p_3 are the three

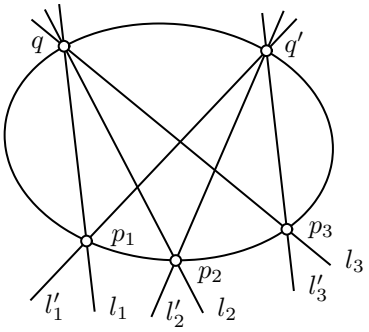


Figure 4.19. A projectivity which leaves a conic and three of its points invariant is the identity transformation.

invariant points, q is an arbitrary fourth point and $q' = Tp$, where T is the P_2 projectivity. According to Section 4.7, T determines a P_1 projectivity between the pencil of lines on q and the pencil of lines on q'. On the other hand, Steiner's theorem states that there is a P_1 projectivity M between the two pencils such that corresponding lines intersect in points on the conic. Since a P_1 projectivity is determined by three pairs of corresponding lines we see that in this case, M is the projectivity induced by T. Consequently, if p is any point on the conic the image of the line pq must be a line on q' which intersects pq in p. Hence, T will map p onto itself. We now have four invariant, non-collinear points on the conic. Since four pair of points determine a P_2 projectivity, T is the identity mapping.

Another way of formulating this is to say that a P_2 projectivity which leaves a proper conic invariant is completely determined by three pairs of corresponding points on the conic: if T and T' both preserve a conic C and both map three points p_1, p_2, p_3 on C onto p'_1, p'_2, p'_3, then $T^{-1}T'$ leaves both the conic and the three points p_1, p_2, p_3 invariant. Hence, $T^{-1}T' = E$ and $T' = T$.

4.8.13 Degenerated conics

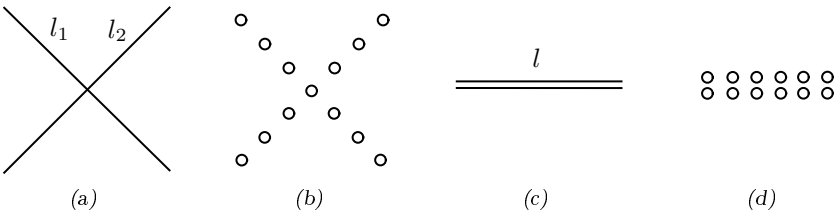
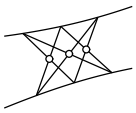


Figure 4.20. Degenerated point conics.

Consider the two lines l_1 and l_2 in Figure 4.20a. The condition that a point p is on either l_1 or l_2 can be written

$$(p^T l_1)(p^T l_2) = 0$$



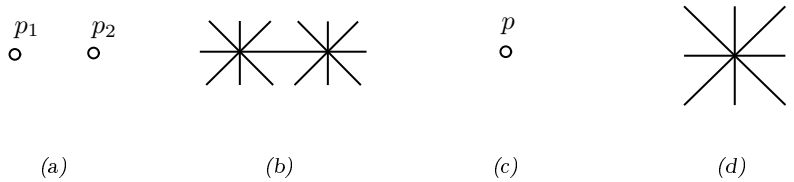


Figure 4.21. Degenerated line conics.

or

$$p^T l_1 l_2^T p = 0 \tag{4.18}$$

Since $l_1 l_2^T$ is a 3×3 matrix, this looks like the equation of a point conic (cf Equation 4.7), but $l_1 l_2^T$ is not symmetric. However,

$$C = l_1 l_2^T + l_2 l_1^T$$

is symmetric. Since $p^T C p = p^T l_1 l_2^T p + p^T l_2 l_1^T p = 2 p^T l_1 l_2^T p$, Equation 4.18 is equivalent to

$$p^T C p = 0$$

Thus, the line pair l_1, l_2 can be seen as a degenerated point conic. It consists of the points that is on either l_1 or l_2, see Figure 4.20b. $p \in N(C)$ if

$$Cp = l_1 l_2^T p + l_2 l_1^T p = 0 \tag{4.19}$$

which is a linear combination of l_1 and l_2. If l_1 and l_2 represent distinct lines, the vectors are linearly independent. In that case (4.19) implies that $l_2^T p = 0$ and $l_1^T p = 0$, which means that p has to be orthogonal to both l_1 and l_2 in R^3. Hence, $\dim N(C) = 1$ and $\operatorname{rank} C = 2$. On the other hand, if $l = (a, b, c)^T$ is a double line (Figure 4.20c) the corresponding conic is

$$C = 2ll^T = 2 \begin{pmatrix} al & bl & cl \end{pmatrix}$$

which has rank 1 and consists of the double points shown in Figure 4.20d. Dually, the two points p_1 and p_2 in Figure 4.21a define the rank 2 line conic

$$C = p_1 p_2^T + p_2 p_1^T$$

shown in Figure 4.21b. The double point p in Figure 4.21c defines the rank 1 line conic shown in Figure 4.21d. To summarize, a rank 2 point conic corresponds to a line pair, a rank 1 point conic to a double line, a rank 2 line conic to a point pair, and a rank 1 line conic to a double point.

In Section 4.8.3, we defined tangents for proper point conics. The same definition works for the degenerated case: the (improper) tangents of a degenerated

point conic are the lines that have 2-contact with the conic. For example, the tangents of the point conic in Figure 4.20b are all the lines through the intersection point of l_1 and l_2, i.e., the line conic in Figure 4.21d. It should be noted though, that there is an infinite number of rank 2 point conics whose tangents form the line conic in Figure 4.21d, namely all point conics defined by two lines intersecting in p. Every line is a tangent of the rank 1 point conic in Figure 4.20d. Dually, the (improper) envelope points of the line conic in Figure 4.21b is the point conic in Figure 4.20d, provided that l is the line through p_1 and p_2. Again, there is an infinite number of rank 2 line conics whose envelope points form the point conic in Figure 4.20d.

We saw that $N(C)$ is non-zero for a degenerated point conic. Since Cp may be zero, it does not necessarily represent a projective line. Therefore, Cp is not always the tangent line through a point p on C.

4.9 Cross-ratio, harmonic sets and separation

Suppose p_1, p_2, p_3 and p_4 are four points in P_1, and that

$$p_i = \begin{pmatrix} p_{i1} \\ p_{i2} \end{pmatrix}, \; i = 1, 2, 3, 4$$

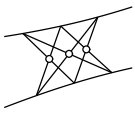
The *cross-ratio* of the four points is

$$(p_1\, p_2 \mid p_3\, p_4) = \frac{\begin{vmatrix} p_{11} & p_{31} \\ p_{12} & p_{32} \end{vmatrix} \cdot \begin{vmatrix} p_{21} & p_{41} \\ p_{22} & p_{42} \end{vmatrix}}{\begin{vmatrix} p_{21} & p_{31} \\ p_{22} & p_{32} \end{vmatrix} \cdot \begin{vmatrix} p_{11} & p_{41} \\ p_{12} & p_{42} \end{vmatrix}} \tag{4.20}$$

Since p_i appears as a column of a determinant in both the numerator and denominator, multiplying p_i by a scalar has no effect on the cross-ratio. Thus, the cross-ratio is well-defined for points in P_1. The denominator becomes zero if p_2 and p_3 coincide or if p_1 and p_4 coincide. We can handle that either by allowing the cross-ratio to be infinite or by treating it as a homogeneous coordinate $(c_1, c_2)^T$ instead of a quotient c_1/c_2, where c_1 is the numerator and c_2 the denominator in (4.20). In any case, the cross-ratio will be undefined if more than two points coincide.

It follows immediately from the definition that

$$\begin{aligned} (p_3\, p_4 \mid p_1\, p_2) &= (p_1\, p_2 \mid p_3\, p_4) \\ (p_2\, p_1 \mid p_3\, p_4) &= (p_1\, p_2 \mid p_4\, p_3) = \frac{1}{(p_1\, p_2 \mid p_3\, p_4)} \\ (p_1\, p_3 \mid p_2\, p_4) &= (p_4\, p_2 \mid p_3\, p_1) = 1 - (p_1\, p_2 \mid p_3\, p_4) \end{aligned}$$



If M is a 2×2 non-singular matrix, then

$$\begin{aligned}(Mp_1\, Mp_2 \mid Mp_3\, Mp_4) &= \frac{\left|Mp_1 \quad Mp_3\right| \cdot \left|Mp_2 \quad Mp_4\right|}{\left|Mp_2 \quad Mp_3\right| \cdot \left|Mp_1 \quad Mp_4\right|} \\ &= \frac{\left|M\begin{pmatrix}p_1 & p_3\end{pmatrix}\right| \cdot \left|M\begin{pmatrix}p_2 & p_4\end{pmatrix}\right|}{\left|M\begin{pmatrix}p_2 & p_3\end{pmatrix}\right| \cdot \left|M\begin{pmatrix}p_1 & p_4\end{pmatrix}\right|} \\ &= \frac{\left|M\right| \cdot \left|p_1 \quad p_3\right| \cdot \left|M\right| \cdot \left|p_2 \quad p_4\right|}{\left|M\right| \cdot \left|p_2 \quad p_3\right| \cdot \left|M\right| \cdot \left|p_1 \quad p_4\right|} = (p_1\, p_2 \mid p_3\, p_4)\end{aligned}$$

Thus, the cross-ratio is invariant under projective transformations. Consequently, if $(p_{i1}, p_{i2})^T$ in (4.20) is interpreted as homogeneous coordinates in a P_1 frame, the cross-ratio given by (4.20) is independent of the choice of frame. In particular, if we choose a frame in which the coordinates of p_1, p_2 and p_3 are $(1,0)^T$, $(0,1)^T$ and $(1,1)^T$ respectively, the cross-ratio becomes

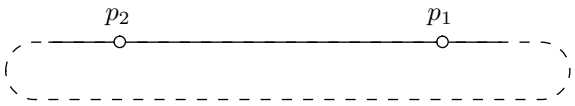
$$(p_1\, p_2 \mid p_3\, p_4) = \frac{\begin{vmatrix}1 & 1\\ 0 & 1\end{vmatrix} \cdot \begin{vmatrix}0 & p_{41}\\ 1 & p_{42}\end{vmatrix}}{\begin{vmatrix}0 & 1\\ 1 & 1\end{vmatrix} \cdot \begin{vmatrix}1 & p_{41}\\ 0 & p_{42}\end{vmatrix}} = \frac{p_{41}}{p_{42}} \tag{4.21}$$

Thus, we can consider the cross-ratio (4.20) as the (non-homogeneous) coordinates of p_4 in a frame in which p_1 is the infinity point, p_2 is the origin and p_3 is the unit point, provided that p_1, p_2 and p_3 are distinct.

So far, we have only defined the cross-ratio for points in P_1. If q_1, q_2, q_3, q_4 are four collinear points in P_2, we can find an embedding A (Section 4.7) such that $q_i = Ap_i,\ i = 1,2,3,4$. We then define $(q_1\, q_2 \mid q_3\, q_4) = (p_1\, p_2 \mid p_3\, p_4)$. Since a different choice of embedding corresponds to a projectivity on P_1 and since the cross-ratio is projectively invariant, $(q_1\, q_2 \mid q_3\, q_4)$ is well-defined.

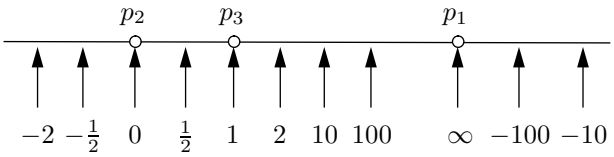
How can $(q_1\, q_2 | q_3\, q_4)$ be computed in practice? Since q_1, q_2, q_3, q_4 are collinear, we can find non-zero scalars λ_1, λ_2, such that $q_3 = \lambda_1 q_1 + \lambda_2 q_2$ (provided that q_1 and q_2 are distinct). Choose the embedding $A = (\lambda_1 q_1, \lambda_2 q_2)$. Then q_1, q_2, q_3 will correspond to $(1,0)^T, (0,1)^T, (1,1)^T$ in P_1. We can easily compute $p_4 = (p_{41}, p_{42})^T$ by solving $Ap_4 = q_4$. From Equation 4.21, we immediately get $(q_1\, q_2 \mid q_3\, q_4) = p_{41}/p_{42}$.

In Euclidean geometry, two points define a line segment. However, a projective line "wraps around" at infinity. In Figure 4.22a it is possible to go from p_1 to p_2 in two directions. There is no way of telling which part of the line the segment p_1, p_2 represents. However, if we add a third point p_3 (Figure 4.22b), we can say (informally) that the segment p_1, p_2, p_3 is the path from p_1 to p_2 that does not contain p_3. In Figure 4.22c the cross-ratio $(p_1\, p_2 \mid p_3\, p_4)$ is shown as a function of the position of p_4. We see that the cross-ratio is negative exactly when p_4 is not on the same segment as p_3. We therefore make the following formal *definitions*: p_1 and p_2 *separate* p_3 and p_4 if $(p_1\, p_2 \mid p_3\, p_4) < 0$. Since the cross-ratio is projectively invariant, the point pairs remain separated even if they are projected to new

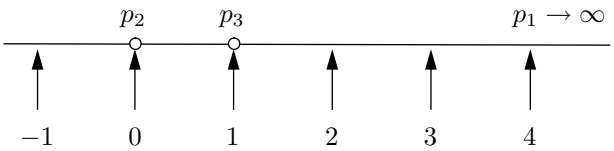


(a) Projectively, the segment p_1p_2 is undefined.

(b) The segment p_1p_2 can be defined as the part that does not contain p_3.

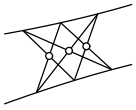


(c) $(p_1\ p_2 \mid p_3\ p_4)$ as a function of the position of p_4.



(d) The scale is changed when p_1 goes to (Euclidean) infinity. When $(p_1\ p_2 \mid p_3\ p_4) = -1$, p_4 is the reflection of p_3 in p_2.

Figure 4.22. The concepts of cross-ratio and separation.



positions on the line. The line segment defined by p_1, p_2, p_3 are the points that is separated from p_3 by p_1, p_2.

When $(p_1\ p_2 \mid p_3\ p_4) = -1$, p_4 is called the *harmonic conjugate* of p_3 with respect to p_1 and p_2, and p_1, p_2, p_3, p_4 is called a *harmonic set*. In Figure 4.22d, we see that with p_1 at (Euclidean) infinity, the harmonic conjugate of p_3 is the reflection of p_3 in p_2.

4.10 Quadrangles and quadrilaterals

Four points p_1, p_2, p_3, p_4 in the projective plane, no three of which are collinear, and the six lines determined by them is called a *quadrangle* (Figure 4.23a). The four points are the *vertices* of the quadrangle and the six lines are its *sides*. Two sides are *opposite* if they do not share a vertex. Opposite sides intersect in the *diagonal points*. Since there are six sides, there are three pairs of opposite sides, and thus three diagonal points. These are called q_1, q_2, q_3 in Figure 4.23b. The triangle $q_1q_2q_3$ is called the *diagonal triangle* of the quadrangle. The sides of this triangle are m_1, m_2, m_3.

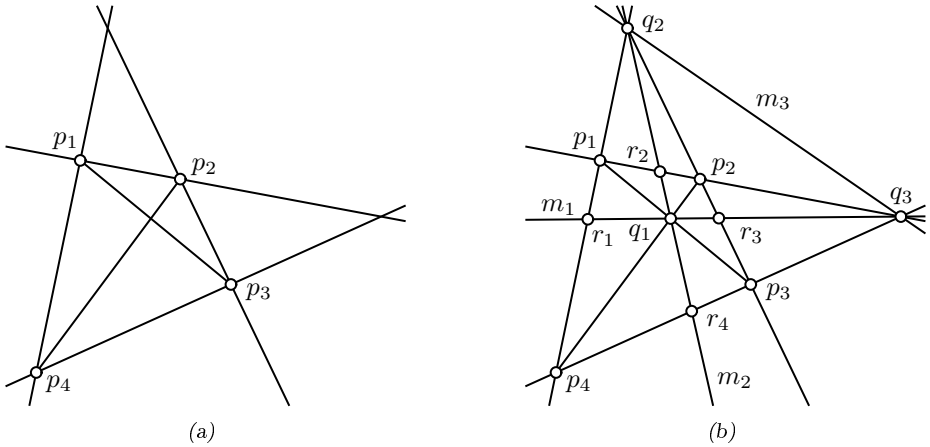


Figure 4.23. A quadrangle.

Each side of the diagonal triangle intersects the sides of the quadrangle in four points: two diagonal points and two other points. In Figure 4.23b, m_1 intersects the quadrangle in r_1, r_3, q_1, q_3 and m_2 intersects the quadrangle in r_2, r_4, q_1, q_2. From q_2, $r_1r_3q_1q_3$ can be projected either onto $p_1p_2r_2q_3$ or $p_4p_3r_4q_3$. From q_1, $p_1p_2r_2q_3$ can also be projected onto $p_3p_4r_4q_3$. Since these projections leave the cross-ratio invariant,

$$\begin{aligned}(r_1\ r_3 \mid q_1\ q_3) = (p_1\ p_2 \mid r_2\ q_3) = (p_4\ p_3 \mid r_4\ q_3) = (p_3\ p_4 \mid r_4\ q_3)\\ = \frac{1}{(p_4\ p_3 \mid r_4\ q_3)} = \frac{1}{(r_1\ r_3 \mid q_1\ q_3)}\end{aligned} \tag{4.22}$$

Thus, $(r_1\ r_3 \mid q_1\ q_3)^2 = 1$. Since the points are distinct, the cross-ratio cannot be 1. Therefore

$$(r_1\ r_3 \mid q_1\ q_3) = -1 \tag{4.23}$$

In other words, the diagonal points are harmonic conjugates with respect to the quadrangle.

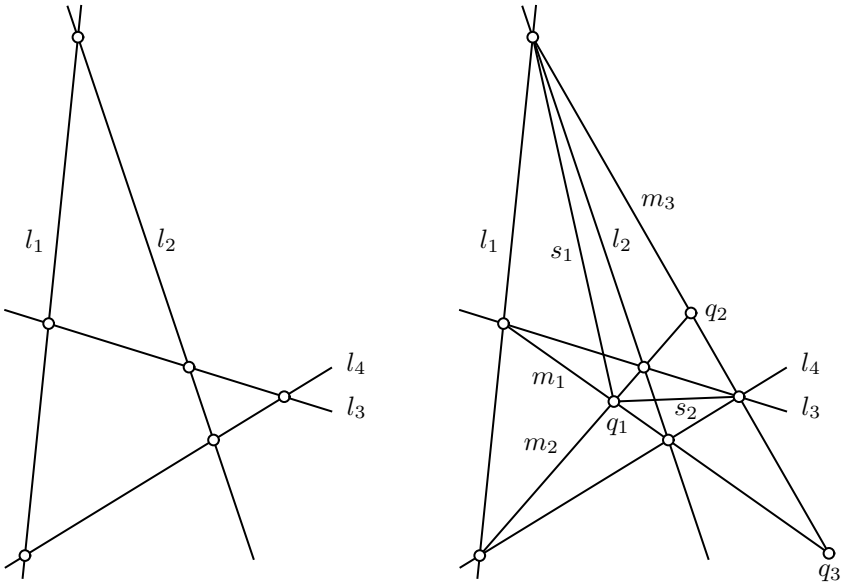


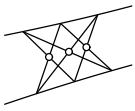
Figure 4.24. A quadrilateral.

The dual of a quadrangle is a *quadrilateral*. It consists of four sides, no three concurrent, and the six vertices where the sides intersect (Figure 4.24a). Two vertices are opposite if they have no side in common. The three lines m_1, m_2, m_3 which join opposite vertices in are the diagonals of the quadrilateral. They form the diagonal triangle whose vertices are q_1, q_2, q_3 (Figure 4.24b).

There are four lines from each vertex of the diagonal triangle to vertices of the quadrilateral. From q_1 in Figure 4.24b, the lines are s_1, s_2, m_1, m_2. The dual of the argument above shows that

$$(s_1\ s_2 \mid m_1\ m_2) = -1$$

Figure 4.25a shows a point p and its polar (Section 4.5 and 4.8.5) with respect to a proper conic C. q is an arbitrary point on the polar and m is the line through p and q. If we assume that neither p nor q is on the conic, the points where m



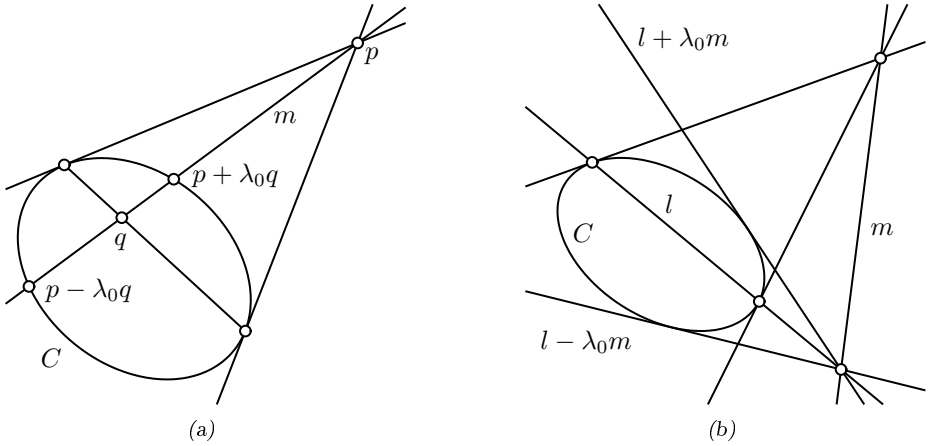


Figure 4.25. Harmonic points and lines with respect to the conic.

intersects the conic can then be written $p + \lambda q$, $\lambda \in \mathrm{R}$. If we insert that into Equation 4.7, we get

$$(p + \lambda q)^T C(p + \lambda q) = q^T Cq\lambda^2 + 2q^T Cp\lambda + p^T Cp = 0$$

Since Cp is the polar of p, $q^T Cp = 0$. Thus, the two roots of the equation can be written $\lambda = \pm\lambda_0$ and the corresponding intersection points are $p + \lambda_0 q$ and $p - \lambda_0 q$. Since p, q, $p + \lambda_0 q$ and $p - \lambda_0 q$ are collinear, they can be mapped to P_1 (Section 4.7). By inserting the P_1 coordinates into Equation 4.20 we obtain

$$(p\, q \mid (p + \lambda_0 q)\, (p - \lambda_0 q)) = -1 \tag{4.24}$$

Thus, every point on the polar of p is a harmonic conjugate of p with respect to the conic. Dually, every line m on the pole of a line l is a harmonic conjugate of l with respect to the conic, see Figure 4.25b.

In Figure 4.26a, a conic has been drawn through the four vertices of the quadrangle from Figure 4.23. From (4.22) and (4.23) we have $(p_1\, p_2 \mid r_2\, q_3) = (p_4\, p_3 \mid r_4\, q_3) = -1$. Combining that with (4.24), we see that both r_2 and r_4 must be on the polar of q_3 and therefore, m_2 is the polar of q_3 with respect to the conic C. Similarly, m_1 must be the polar of q_2, and since q_1 is on the polars of both q_2 and q_3, it must be the pole of m_3. The triangle $q_1q_2q_3$ is therefore *self-polar* with respect to the conic. The argument is valid for any conic on p_1, p_2, p_3, p_4. Thus, we have showed that the diagonal triangle of a quadrangle is self-polar with respect to the pencil of conics determined by the four vertices of the quadrangle (Figure 4.26b). Dually, the diagonal triangle of a quadrilateral is self-polar with respect to any conic touching the four sides of the quadrilateral (Figure 4.27).

Returning to Figure 4.26a, consider the tangents t_1, t_2, t_3, t_4 through the vertices p_1, p_2, p_3, p_4 of the quadrangle. Let $u_1, u_2, u_3, u_4, u_5, u_6$ be the six vertices of the quadrilateral defined by t_1, t_2, t_3, t_4, see Figure 4.28. Obviously, the line p_1p_4

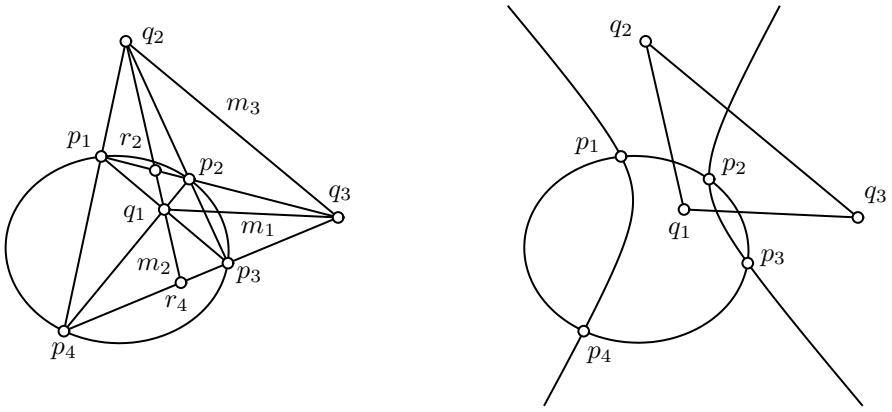


Figure 4.26. A self-polar triangle.

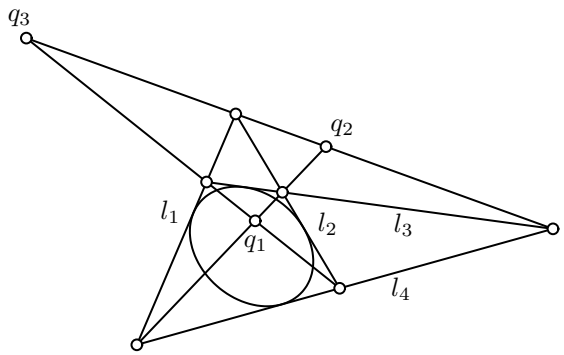
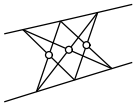


Figure 4.27. The diagonal triangle $q_1q_2q_3$ is self-polar with respect to the quadrilateral $l_1l_2l_3l_4$.



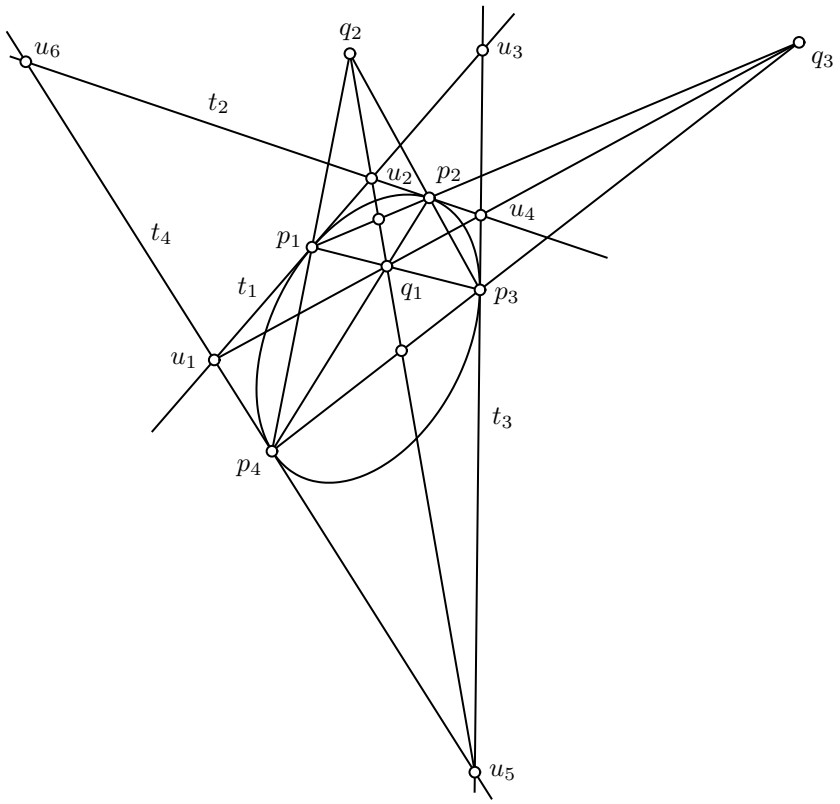


Figure 4.28. A quadrilateral and a quadrangle with the same self-polar diagonal triangle.

is the polar of u_1 and the line p_2p_3 is the polar of u_4. Since q_2 is on both these lines, it must be pole of u_1u_4. But we saw above that q_1q_3 is the polar of q_2. Thus, u_1u_4 and q_1q_3 is the same line. Similarly, u_2u_5 is the same line as q_1q_2 and u_3u_6 the same line as q_2q_3. But u_1u_4, u_2u_5, u_3u_6 are the diagonals of the quadrilateral t_1, t_2, t_3, t_4, and q_1, q_2, q_3 the diagonal points of the quadrangle p_1, p_2, p_3, p_4. Hence, the quadrilateral and the quadrangle have the same (self-polar) diagonal triangle.

Furthermore, if we consider u_1, u_2, u_4, u_5 as the vertices of a quadrangle, we already know that the intersection of u_1u_4 and u_2u_5 is the pole of u_3u_6 *with respect to the conic through* u_1, u_2, u_4, u_5. But we have just seen that u_1u_4 and u_2u_5 intersect in q_1 and that u_3u_6 is the same line as q_2q_3. Thus, q_1 is the pole of q_2q_3 both with respect to the pencil of conics through p_1, p_2, p_3, p_4 and with respect to the pencil of conics through u_1, u_2, u_4, u_5.

4.11 Metrics

A *metric* in P_2 specifies how the angle between two lines or the distance between two points is measured. From angles and distances, other metric concepts can be derived, such as area, midpoints and angle bisectors. None of these concepts are meaningful if it is not clear what metric we are referring to.

Metrics can be defined in several ways. We will follow [Klein28, Winger62] and define a metric in terms of its *absolute elements*, since that definition is geometrically intuitive and clearly shows the relationship between different metric geometries.

4.11.1 Measuring distances and angles

The absolute elements associated with a metric are a (possibly degenerated) point conic Ω for measuring distances and a (possibly degenerated) line conic Ψ for measuring angles. If Ω and Ψ are proper they should be related by $\Psi = \Omega^{-1}$ (cf Section 4.8.4). In that case, Ψ consists of the tangents of Ω (Figure 4.14, page 39). If $\operatorname{rank}\Omega = 2$, Ψ consists of the (improper) tangents of Ω (Section 4.8.13). Dually, if $\operatorname{rank}\Psi = 2$, Ω consists of the envelope points of Ψ. When Ω and Ψ are proper, they carry the same information and we can talk about *the* absolute conic. However, it is important to make the distinction between point and line conics if they are degenerated. For example, an absolute point conic of rank 1 does not uniquely determine the line conic we need for measuring angles.

The points on Ω are called *ideal points*, and the lines of Ψ (which are also tangents of Ω) are called *ideal lines*.

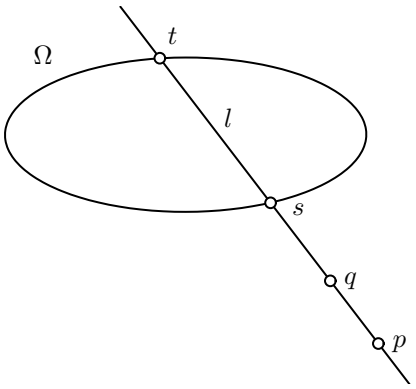
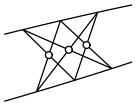


Figure 4.29. Measuring the distance between p and q.

Figure 4.29 shows two points p and q on a line l. s and t are the ideal points on l. We define the distance between p and q as

$$\operatorname{dist} pq = k_1 \ln (s\, t \mid p\, q) \tag{4.25}$$



where k_1 is a constant we choose. If the cross-ratio is complex, ln represents the complex logarithm: $\ln z = \ln|z| + i \arg z$. $\arg z$ is only determined up to a multiple of 2π. In (4.25), we use the principal value $0 \leq \arg z < 2\pi$.

If the points p and q are real, so is the line l. If the conic has real coefficients, the ideal points s and t will then either be real, in which case the distance measure is called *hyperbolic*, or conjugate complex, in which case the distance measure is called *elliptic*. When the distance measure is elliptic, we see from Equation 4.20 that since $t = \overline{s}$ and p, q are real

$$|(s\ t \mid p\ q)| = |\frac{r_1 e^{\varphi_1} \cdot r_2 e^{\varphi_2}}{r_1 e^{-\varphi_1} \cdot r_2 e^{-\varphi_2}}| = 1$$

and therefore

$$\operatorname{dist} pq = k_1 \ln (s\ t \mid p\ q) = k_1 (\ln 1 + i \arg (s\ t \mid p\ q)) = k_1 i \arg (s\ t \mid p\ q)$$

Thus, elliptic distances are bounded. On the other hand, if the distance measure is hyperbolic, s and t are real, $|(s\ t \mid p\ q)| \neq 1$ and distances are unbounded.

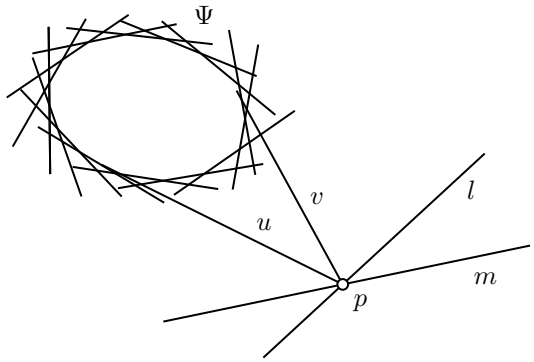


Figure 4.30. Measuring the angle between l and m.

Angles are given the dual definition, as shown in Figure 4.30. If l and m are two lines intersecting in a point p not on the envelope of Ψ, and u and v are the ideal lines through p, we define the angle between l and m as

$$\operatorname{ang} lm = k_2 \ln (u\ v \mid l\ m) \tag{4.26}$$

Again, k_2 is a constant we choose. The angle measure is called hyperbolic or elliptic depending on whether u and v are real or conjugate complex. Elliptic angles are bounded, hyperbolic angles unbounded.

Note that it is not meaningful to classify the distance between two points as elliptic or hyperbolic unless the two points are real. Similarly, when we talk about elliptic and hyperbolic angles, we assume that the two lines are real. Furthermore, the coefficient matrix of Ω must be real, although Ω does not necessarily contain any real points.

4.11.2 Degenerated geometries

The distance measure (4.25) works fine even if Ω degenerates into a line pair (a rank 2 conic), see Figure 4.31. However, if Ω degenerates to a double line (a rank 1 conic), the situation gets more complicated. In this case, s and t become a double point as shown in Figure 4.32. If k_1 is finite in Equation 4.25, $\operatorname{dist} pq \equiv 0$ since $\ln(t\ t \mid p\ q) = \ln 1 = 0$ for all points p and q. Thus, with a finite k_1 the distance measure defined by Equation 4.25 will not be meaningful. Nevertheless, (4.25) can still be used if we let k_1 depend on the shape of the absolute conic as it degenerates. To see that, choose a frame in which

$$\Omega = \begin{pmatrix} \varepsilon & 0 & 0 \\ 0 & \varepsilon & 0 \\ 0 & 0 & -1 \end{pmatrix}, p = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}, q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

Provided that q is not on Ω, the ideal points can be written $p + \lambda q$, where $\lambda \in \mathrm{R}$ is such that

$$(p + \lambda q)^T \Omega (p + \lambda q) = q^T \Omega q \lambda^2 + 2p^T \Omega q \lambda + p^T \Omega p = 0$$

With $\Omega_{pq} = p^T \Omega q$ and $\Delta_{pq} = \Omega_{pq}^2 - \Omega_{pp}\Omega_{qq}$ we get

$$\lambda = \frac{-\Omega_{pq} \pm \sqrt{\Delta_{pq}}}{\Omega_{qq}}$$

Let λ_1 and λ_2 be the two roots. Choose an embedding of P_1 (Section 4.7) such that $(0,1)^T \mapsto p$ and $(1,0)^T \mapsto q$. Then it is easily seen that

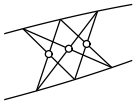
$$((p + \lambda_1 q)\ (p + \lambda_2 q) \mid p\ q) = \frac{\lambda_1}{\lambda_2} = \frac{-\Omega_{pq} + \sqrt{\Delta_{pq}}}{-\Omega_{pq} - \sqrt{\Delta_{pq}}} = 1 + \frac{2\sqrt{\Delta_{pq}}}{-\Omega_{pq} - \sqrt{\Delta_{pq}}}$$

This is known as Cayley-Klein's distance formula. If we expand the discriminant Δ_{pq} it turns out that the terms not containing ε are cancelled out:

$$\begin{aligned} \Delta_{pq} &= (\varepsilon p_1 q_1 + \varepsilon p_2 q_2 - p_3 q_3)^2 - (\varepsilon p_1^2 + \varepsilon p_2^2 - p_3^2)(\varepsilon q_1^2 + \varepsilon q_2^2 - q_3^2) \\ &= (p_1^2 q_3^2 + p_3^2 q_1^2 - 2p_1 q_3 p_3 q_1 + p_2^2 q_3^2 + p_3^2 q_2^2 - 2p_2 q_3 p_3 q_2)\varepsilon + O(\varepsilon^2) \\ &= ((p_1 q_3 - p_3 q_1)^2 + (p_2 q_3 - p_3 q_2)^2)\varepsilon + O(\varepsilon^2) \end{aligned}$$

Thus, $\sqrt{\Delta_{pq}} = O(\sqrt{\varepsilon}) \to 0$ and $-\Omega_{pq} - \sqrt{\Delta_{pq}} \to p_3 q_3$ as $\varepsilon \to 0$. If $p_3 q_3 \neq 0$, the Taylor expansion of the distance measure is

$$\operatorname{dist} pq = k_1 \ln\left(1 + \frac{2\sqrt{\Delta_{pq}}}{-\Omega_{pq} - \sqrt{\Delta_{pq}}}\right) = k_1 \left(\frac{2\sqrt{\Delta_{pq}}}{-\Omega_{pq} - \sqrt{\Delta_{pq}}} + O(\varepsilon)\right)$$



If we want this to converge to something other than zero, we must choose $k_1 = k_1'/\sqrt{\varepsilon}$, where k_1' is a fixed and finite constant. Then

$$\operatorname{dist} pq = \frac{k_1'}{\sqrt{\varepsilon}} \cdot \frac{2\sqrt{\Delta_{pq}}}{-\Omega_{pq} - \sqrt{\Delta_{pq}}} + O(\sqrt{\varepsilon})$$
$$\to 2k_1' \frac{\sqrt{(p_1q_3 - p_3q_1)^2 + (p_2q_3 - p_3q_2)^2}}{p_3q_3} = 2k_1' \sqrt{(\frac{p_1}{p_3} - \frac{q_1}{q_3})^2 + (\frac{p_2}{p_3} - \frac{q_2}{q_3})^2}$$

But $(p_1/p_3, p_2/p_3)$ and $(q_1/q_3, q_2/q_3)$ are the Euclidean coordinates of p and q with the standard embedding of the Euclidean plane. Thus, $\operatorname{dist} pq$ is the usual Euclidean distance where the constant k_1' determines the unit length. The assumption $p_3q_3 \neq 0$ which we made above means that p and q must not be on the Euclidean line at infinity if we want to measure the distance between them.

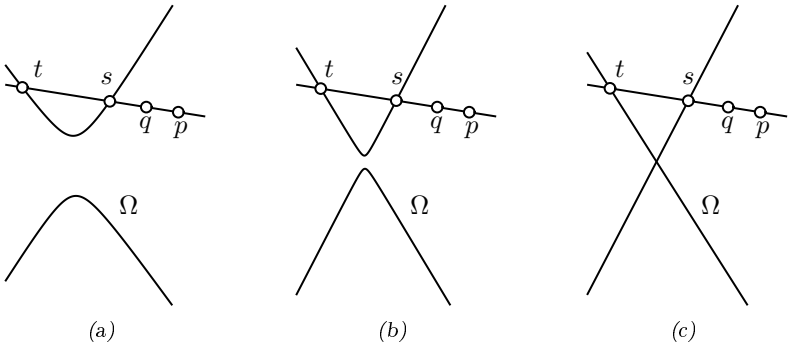


Figure 4.31. The absolute point conic degenerates to a line pair (rank 2).

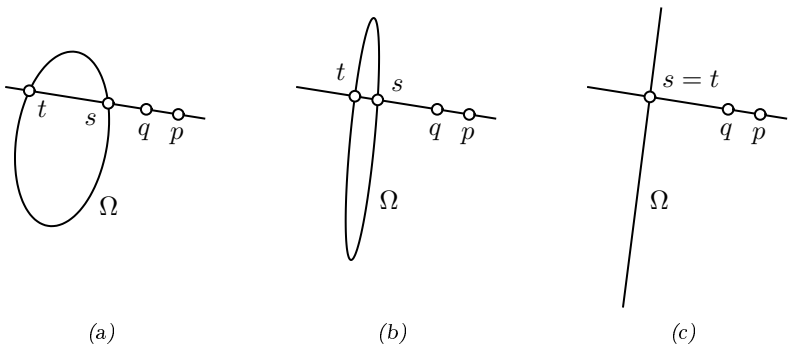


Figure 4.32. The absolute point conic degenerates to double line (rank 1).

This distance measure, which resulted from the collapse of Ω, is called *parabolic*. It is the limiting case between the hyperbolic ($\varepsilon > 0$) and elliptic ($\varepsilon < 0$) distance measures. Thus, distances in the Euclidean plane are parabolic.

The absolute line conic Ψ can collapse in a similar way, which would result in a parabolic angle measure. Since there are three types of distances measures and three types of angle measures, there are nine possible combinations. By tradition, however, one usually assumes that the angle is bounded, i.e., that the angle measure is elliptic. The three remaining combinations are studied in *hyperbolic geometry*, *elliptic geometry* and *parabolic geometry* (or Euclidean geometry). Thus, the name of a geometry specifies the type of *distance* measure used.

4.11.3 Parallel and perpendicular lines

Two lines are *parallel* if they intersect in a point on Ω. In Figure 4.33a l is parallel to both m_1 and m_2. If Ω is proper or of rank 2, l usually intersects Ω in two distinct points. Therefore, $l \parallel m_1$, $l \parallel m_2$ does not imply $m_1 \parallel m_2$. However, in Euclidean geometry, Ω is a double line (the line at infinity), and parallelism is transitive (Figure 4.33b).

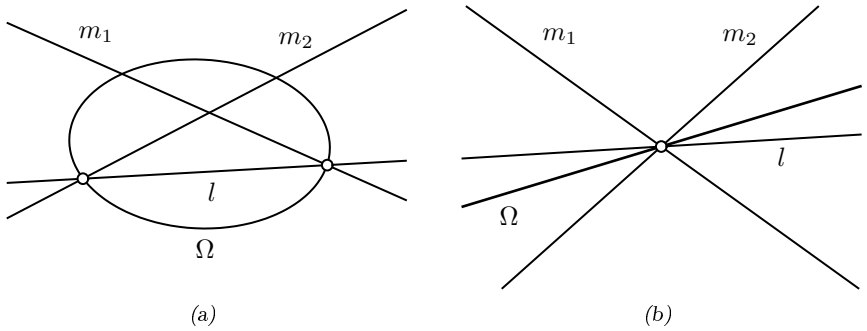


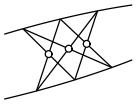
Figure 4.33. Parallel lines in a non-degenerated geometry (a), and in Euclidean geometry (b).

Two lines are *perpendicular* (or orthogonal) if they are harmonic conjugates with respect to the absolute conic. To be more specific, if l and m intersect in a point p and u, v are the ideal lines through p,

$$l \perp m \;\Leftrightarrow\; (u\, v \,|\, l\, m) = -1$$

(Note that perpendicularity and parallelism are not dual properties.)

In Section 4.9 we saw that the pencil of lines on the pole of a given line l with respect to a conic C are the harmonic conjugates of l with respect to C. Thus, if Ω is proper, all lines which are perpendicular to l intersect in the pole of l with respect to Ω. A consequence of this is that two lines always have a common perpendicular, namely the line through their absolute polars. If the lines are parallel, their common perpendicular is tangent to Ω at their point of intersection. Note that this will not be true if Ω is degenerated. In Euclidean geometry for example, two intersecting lines have no common perpendicular, and two parallel lines have an infinite number of perpendiculars.



The constant k_2 in Equation 4.26 is usually chosen so that the angle between perpendicular lines are $\pi/2$. Using Equation 4.26 we get

$$k_2 \ln(-1) = k_2 i\pi = \frac{\pi}{2} \;\Rightarrow\; k_2 = \frac{1}{2i}$$

4.11.4 Circles

A circle is usually thought of as a Euclidean concept. However, we can define a circle C as the locus of a point with constant distance r (the radius) to a fixed point c (the center).

$$\{p\colon \operatorname{dist} pc = r\}$$

It can be shown that C is a conic which has double contact with Ω [Winger62]. Furthermore, if q_1 and q_2 are the two points of contact, the chord q_1q_2 is the polar of c with respect to Ω. Dually, the envelope of lines with a certain angle to a fixed line is the line conic which has double contact with Ψ.

4.11.5 Isometries

An *isometry* is a projectivity that does not affect distances or angles. Thus, M is an isometry if $\operatorname{dist}(Mp, Mq) = \operatorname{dist}(p, q)$ and $\operatorname{ang}(M^{-T}l, M^{-T}m) = \operatorname{ang}(l, m)$ for all points p, q and all lines l, m. If Ω is proper, the isometries are the projectivities that map Ω onto itself. That follows immediately from the definition of distances and angles and the fact that the cross-ratio is projectively invariant. It is also apparent that the isometries form a subgroup of the full matrix group. However, it is not so easy to characterize the isometries when the absolute conic is degenerated. A parabolic measure is determined not only by the absolute conic but also by the limiting process which degenerated it. See Section 4.12.3 for a characterization of Euclidean isometries.

4.12 Special geometries

4.12.1 Hyperbolic geometry

In hyperbolic geometry, the distance measure is by definition hyperbolic and the angle measure elliptic. Thus, each line must contain two real and distinct ideal points, and on each point there must be two conjugate complex ideal lines.

For any line to contain real and distinct ideal points, the absolute point conic Ω must be proper and contain an infinite number of real points. Such a conic divides the real projective plane into interior and exterior points (Section 4.8.6). In hyperbolic geometry one usually considers only the interior points of Ω since only those points are incident on two conjugate complex ideal lines. To emphasize that, the interior points are called *ordinary* and the exterior ones *ultraideal* in hyperbolic geometry. Ordinary lines are the ones containing an ordinary point.

Other lines are either ideal or ultraideal. Since every ordinary line contains two real ideal points, the distance between two ordinary points will be hyperbolic, as required. If we want the distance measure to be real, we must choose a real value for k_1 since the cross-ratio in Equation 4.25 will be real.

Strictly speaking, the hyperbolic plane consists solely of the ordinary points and lines. Just as the line at infinity is not part of the Euclidean plane, the ideal and ultraideal points and lines are not part of the hyperbolic plane. Therefore, two lines in the projective plane that intersect in an ideal or ultraideal point (with respect to Ω) do not really intersect at all in the hyperbolic plane. If we want to make a concept such as parallelism intrinsic to hyperbolic geometry, we cannot define parallel lines as lines intersecting in an ideal point (as we did in Section 4.11). Instead, we can choose the following definition, analogous to the definition of parallel lines in Euclidean geometry: The lines through a given point p either intersect or do not intersect a given line l. Two lines in the pencil of lines on p will separate the intersectors from the non-intersectors. They are the *parallels* of l through p. Needless to say, such definitions are very cumbersome. Therefore, we will always treat the hyperbolic plane as embedded in P_2. That will allow us to use ideal and ultraideal elements in our definitions. However, it is important to realize that the distance measure is guaranteed to be hyperbolic and the angle measure is guaranteed to be elliptic only for ordinary points and lines. The distance between two real ultraideal points may not even be real.

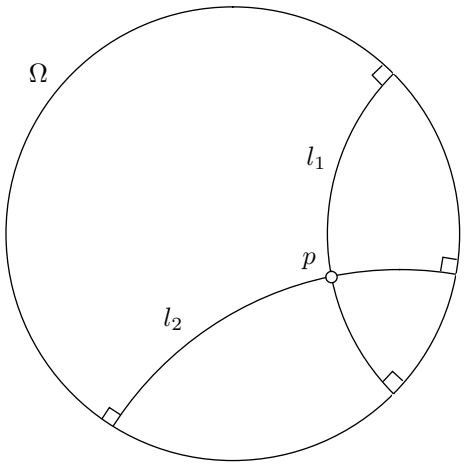
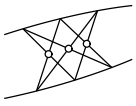


Figure 4.34. Two lines and a point drawn on a Poincaré disc.

A common way of visualizing the hyperbolic plane is to draw Ω as a circle and the ordinary lines as circles perpendicular to Ω, see Figure 4.34. Such a representation is called a *Poincaré disc* and can be constructed in R^3 as shown in Figure 4.35 [Klein28]. We assume here that Ω is the unit circle[2] Let p be an

[2] If it is not, we will first have to find a projectivity which maps Ω to the unit circle and apply that to every point in the plane.



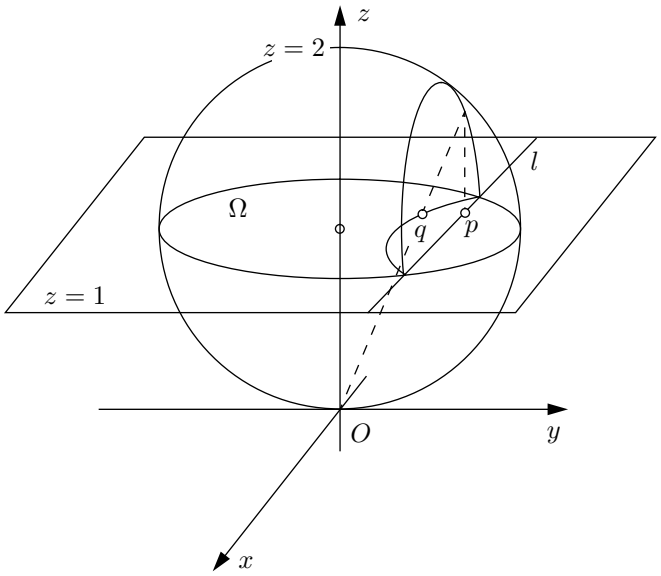


Figure 4.35. Mapping points in the projective plane onto the Poincaré disc.

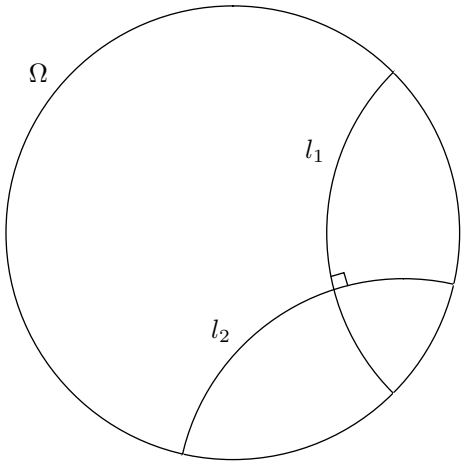


Figure 4.36. The Euclidean angle between the arcs on the disc equals the actual angle (defined by Ω) between the lines.

ordinary point in the hyperbolic plane, i.e. an interior point of Ω. Project p vertically onto the upper half of a sphere of radius 1 centered at $z = 1$. Then project that point back onto the plane $z = 1$ from O, which is the south pole of the sphere. Obviously, Ω will be mapped onto itself. The points of a projective line l will be mapped onto a circle segment in $z = 1$, and that segment will intersects Ω orthogonally. Furthermore, the angle (as defined by Equation 4.26) between two ordinary, intersecting lines in the hyperbolic plane will be the same as the *Euclidean* angle between the circular arcs which represent them on the Poincaré disc (Figure 4.36).

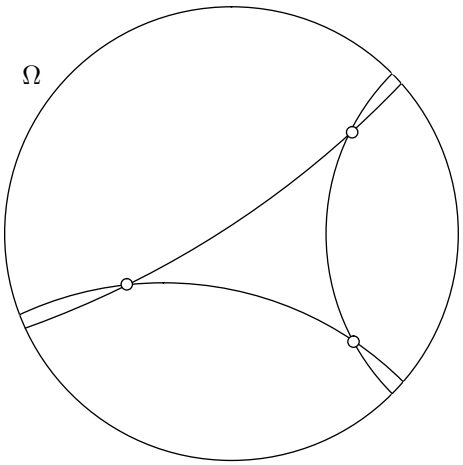
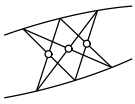


Figure 4.37. The hyperbolic angle sum of a triangle is less than π.

The Poincaré disc makes it easier to understand some aspects of hyperbolic geometry that has to do with angles. For example, it is obvious from Figure 4.37 that the sum of the interior angles of a triangle is less than π. In fact, as the three vertices approaches Ω, the sum of the angles approaches zero.

4.12.2 Elliptic geometry

In elliptic geometry, both the distance measure and the angle measure are elliptic. The absolute point conic is proper but contains no real points. Every (real) line contains two conjugate complex ideal points and every point is on two conjugate complex ideal lines. Thus, the elliptic and projective planes contain the same points and lines. There is no distinction between ordinary and ultraideal points and lines as in hyperbolic geometry. The constant k_1 in Equation 4.25 is usually chosen to be imaginary so that the distance between two real points will be real.



4.12.3 Euclidean geometry

In Euclidean geometry, the absolute line conic is

$$\Psi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

which has rank 2. If $l = (l_1, l_2, l_3)^T$ is a line on Ψ, $l_1^2 + l_2^2 = 0$. There is only one real line that satisfies this condition, namely $(0, 0, 1)^T$ which is the Euclidean line at infinity (Section 4.1). However, there is an infinite number of complex lines on Ψ. In fact, Ψ consists of all lines that are on either $I = (1, i, 0)^T$ or $J = (1, -i, 0)^T$ since

$$IJ^T + JI^T = 2\Psi$$

(cf Section 4.8.13). A real point $p = (p_1, p_2, p_3)^T$ is on the ideal, conjugate complex lines $u = p \times I$ and $v = p \times J$. Thus, the angle measure is elliptic. p is an (improper) envelope point of Ψ only if $u = v$. That happens when p is on $u \times v$, i.e., when $p_3 = 0$. Hence, the envelope points of Ψ are exactly the points on

$$\Omega = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We saw in Section 4.11 that this absolute point conic through a limiting process gave rise to the usual Euclidean distance measure. We must now verify that Equation 4.26 actually represents the usual Euclidean angle, for the choice of Ψ above. Suppose l and m are two lines through the origin. Let the angles between the lines and the x-axis be α and β respectively (Figure 4.38). The line equations in Euclidean coordinates are $y = x \tan\alpha$ and $y = x \tan\beta$. The ideal lines through the origin are $u = (-i, 1, 0)^T$ and $v = (i, 1, 0)^T$. These four lines intersect the line $x = 1$ in four points:

$$p = \begin{pmatrix} 1 \\ \tan\alpha \\ 1 \end{pmatrix},\ q = \begin{pmatrix} 1 \\ \tan\beta \\ 1 \end{pmatrix},\ s = \begin{pmatrix} 1 \\ i \\ 1 \end{pmatrix},\ t = \begin{pmatrix} 1 \\ -i \\ 1 \end{pmatrix}$$

With the embedding

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

these points correspond to

$$\begin{pmatrix} 1 \\ \tan\alpha \end{pmatrix}, \begin{pmatrix} 1 \\ \tan\beta \end{pmatrix}, \begin{pmatrix} 1 \\ i \end{pmatrix}, \begin{pmatrix} 1 \\ -i \end{pmatrix}$$

Thus

$$(u\,v \mid l\,m) = (s\,t \mid p\,q) = \frac{\begin{vmatrix}1 & 1\\ i & \tan\alpha\end{vmatrix} \cdot \begin{vmatrix}1 & 1\\ -i & \tan\beta\end{vmatrix}}{\begin{vmatrix}1 & 1\\ i & \tan\beta\end{vmatrix} \cdot \begin{vmatrix}1 & 1\\ -i & \tan\alpha\end{vmatrix}} = \frac{(\tan\alpha - i)(\tan\beta + i)}{(\tan\beta - i)(\tan\alpha + i)}$$
$$= \frac{(\cos\alpha + i\sin\alpha)(\cos\beta - i\sin\beta)}{(\cos\beta + i\sin\beta)(\cos\alpha - i\sin\alpha)} = \frac{e^{i\alpha}e^{-i\beta}}{e^{i\beta}e^{-i\alpha}} = e^{2i(\alpha-\beta)}$$

This is known as Laguerre's angle formula. Equation 4.26 then gives us (with $k_2 = 1/2i$)

$$\operatorname{ang} lm = \frac{1}{2i}\ln e^{2i(\alpha-\beta)} = \alpha - \beta$$

If l and m instead intersect in $(a, b, 1)^T$, we first apply the projectivity

$$T = \begin{pmatrix}1 & 0 & -a\\ 0 & 1 & -b\\ 0 & 0 & 1\end{pmatrix}$$

T is called a *translation* since

$$T\begin{pmatrix}x\\ y\\ 1\end{pmatrix} = \begin{pmatrix}x-a\\ y-b\\ 1\end{pmatrix}$$

It is easy to verify that T maps Ψ onto itself: $T\Psi T^T = \Psi$. Since T affects neither the usual Euclidean angle nor the angle measure (4.26), $\operatorname{ang} lm = \alpha - \beta$ for lines intersecting in an arbitrary point.

We just saw that angles are invariant under translations. Actually, a translation is a Euclidean isometry since it also preserves distances. If $p = (p_1, p_2, 1)^T$ and $q = (q_1, q_2, 1)^T$

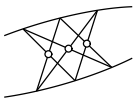
$$\begin{aligned}\operatorname{dist}(Tp, Tq) &= \sqrt{(p_1 - a - (q_1 - a))^2 + (p_2 - b - (q_2 - b))^2}\\ &= \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} = \operatorname{dist}(p, q)\end{aligned}$$

It is also easy to verify that the projectivity

$$R = \begin{pmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{pmatrix}$$

which represents a *rotation* by θ around the origin is an isometry. A combination of translations and rotations is called a *rigid motion*. The projectivity

$$M = \begin{pmatrix}-1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$



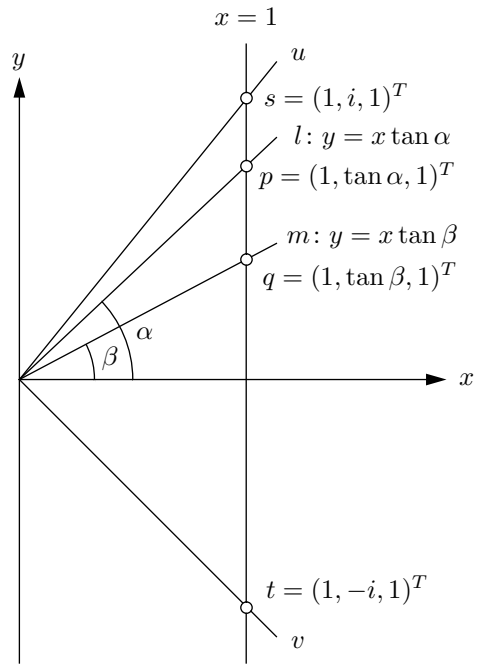


Figure 4.38. Verifying that I and J defines Euclidean angles.

is called a *reflection* since it maps $(x, y, 1)^T$ to $(-x, y, 1)^T$. Since M swaps I and J, and since $(u\, v \mid l\, m) = 1/(v\, u \mid l\, m)$ and $\ln(1/c) = -\ln c$, it follows that M reverses all angles. However, provided that we are not considering *directed* angles and distances, M is an isometry. Translations, rotations and reflections together form the largest subgroup of isometries in Euclidean geometry.

The projectivity

$$S = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where $|s| \neq 1$ scales a geometric figure. Since $S\Psi S^T = s^2\Psi$, S preserves angles. However, S is not an isometry since distances obviously are affected by scaling. Interestingly, $S^{-T}\Omega S^{-1} = \Omega$. We conclude that if $\operatorname{rank}\Omega = 1$ (i.e, if the distance measure is parabolic) a projectivity S which maps Ω onto itself does not necessarily preserve distances.

A combination of rotation, translation, reflection and scaling is called an angle preserving transformation or *similarity transformation*. A similarity transformation leaves I and J invariant. There is no corresponding concept in non-degenerated geometries. There, a projectivity either leaves distances and angles invariant, in which case it is an isometry, or it affects both.

In Section 4.11.4, we mentioned that a circle (the locus of a point at constant

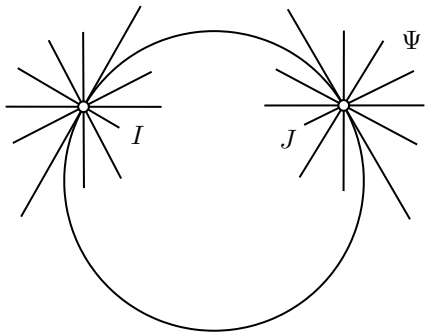


Figure 4.39. A Euclidean circle is on I and J.

distance from a fixed point) is a conic which has contact with the absolute conic in two points. In the degenerated Euclidean case, that would correspond to a conic through I and J since such a conic would be tangent to Ψ in those points, see Figure 4.39. Let us verify this. If C is the coefficient matrix from Equation 4.6, page 36

$$I^T C I = a - b + 2ic = 0 \Rightarrow \begin{cases} a = b \\ c = 0 \end{cases}$$

If C is scaled so that $a = b = 1$, the point equation becomes $x^2+y^2+2dx+2ey+f = 0$. By substituting $f = d^2 + e^2 - r^2$ we get

$$(x + d)^2 + (y + e)^2 = r^2$$

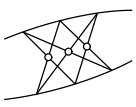
which is the familiar equation for a circle with radius r centered at $(-d, -e)$. Because I and J are on all Euclidean circles, they are often called the *circular points*.

The *center* of a conic is the pole of the line at infinity. For a parabola, which is tangent to the line at infinity, the center is the point of contact.

In Figure 4.40a, the ideal lines that are tangent to a conic C has been drawn. The points r_1, r_2, r_3, r_4 in which the lines intersect are the *focal points* of the conic. Provided that C is real, two of them will be real and two will be conjugate complex. To see that, consider Figure 4.40b where l is the line at infinity, p is the center of the conic (the pole of l), m is the line pI, and m' is the line pJ. l and p are real while m and m' are conjugate complex. The situation is the same as in Figure 4.25b, page 55 although in this case, l is an exterior line. The tangents through I can therefore be written $l + \lambda m$ and $l - \lambda m$ where

$$\lambda^2 = -\frac{l^T C^{-1} l}{m^T C^{-1} m} \tag{4.27}$$

Similarly, the tangents through J are $l + \lambda' m'$ and $l - \lambda' m'$. It is easy to see that



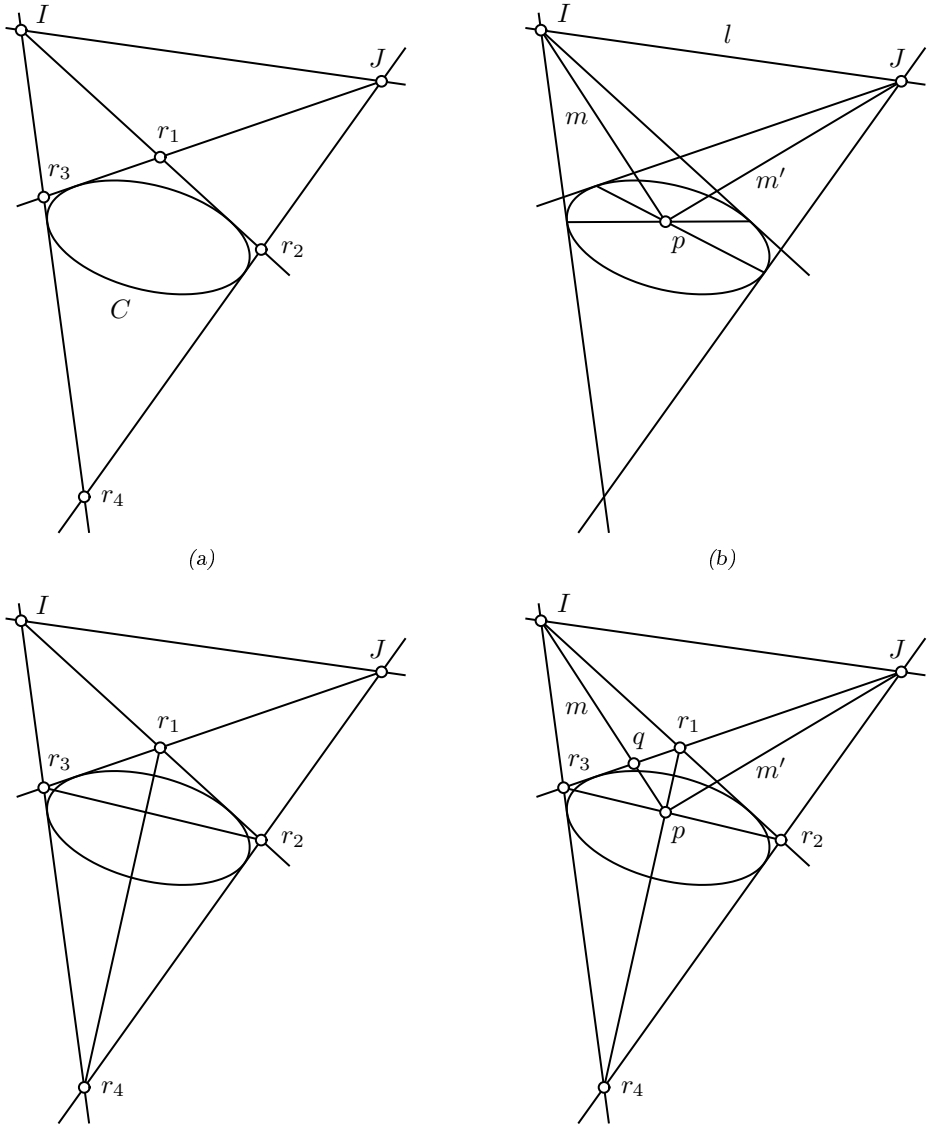


Figure 4.40. The focal points and axes of a conic.

$m' = \overline{m}$ and $\lambda' = \pm\overline{\lambda}$. Assuming $\lambda' = \overline{\lambda}$, the focal points are

$$\begin{aligned} r_1 &= (l + \lambda m) \times (l + \overline{\lambda}\overline{m}) \\ r_2 &= (l + \lambda m) \times (l - \overline{\lambda}\overline{m}) \\ r_3 &= (l - \lambda m) \times (l + \overline{\lambda}\overline{m}) \\ r_4 &= (l - \lambda m) \times (l - \overline{\lambda}\overline{m}) \end{aligned} \tag{4.28}$$

Since l is real, $\overline{r_1} = -r_1$, $\overline{r_4} = -r_4$, and $\overline{r_2} = -r_3$. Therefore, projectively, r_1 and r_4 represent real points while r_2 and r_3 are conjugate complex.

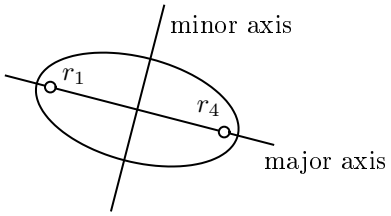


Figure 4.41. The true focal points and axes of a conic.

The line r_1r_4 is the *major axis* and r_2r_3 is the *minor axis* of the conic, see Figure 4.40c. Note that also the minor axis is real since r_2 and r_3 are conjugate complex. The axes are diagonals of the quadrilateral formed by the four tangents. Therefore, they intersect in p, the pole of l (cf Figure 4.27, page 56). In other words, the axes intersect in the center of the conic. Furthermore, if we compare Figure 4.40d with Figure 4.23b on page 53 where p_1, p_2, r_2, q_3 were found to be harmonic, we see that r_3, r_1, q, J are harmonic (q is the intersection of m and r_1r_3). It follows that m, m' and the major and minor axes are harmonic. Hence, the axes are perpendicular. Of course, since the line at infinity is drawn as an ordinary line in Figure 4.40c, the location of the focal points and axes in the figure is not correct. Figure 4.41 shows the same drawing with I and J at their proper positions.

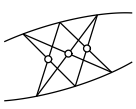
Confocal conics have the same focal points. From the definition above it is clear that all conics tangent to the same four ideal lines are confocal, see Figure 4.42.

The equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{4.29}$$

where $0 < b < a$ represents an ellipse centered at the origin, whose major axis is $2a$ and minor axis $2b$. Using Equations 4.27 and 4.28 we can compute the focal points:

$$\begin{pmatrix} \sqrt{a^2 - b^2} \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} -\sqrt{a^2 - b^2} \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ i\sqrt{a^2 - b^2} \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -i\sqrt{a^2 - b^2} \\ 1 \end{pmatrix}$$



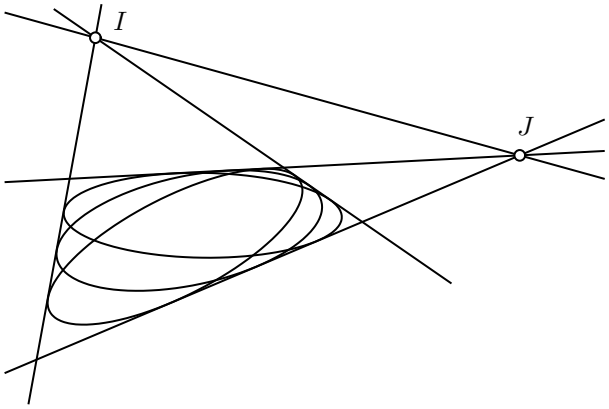


Figure 4.42. Confocal conics have the same ideal tangents.

The sum of the distances from an arbitrary point $p = (x, y)$ on the ellipse to the two real focal points is

$$d = \sqrt{(x - \sqrt{a^2 - b^2})^2 + y^2} + \sqrt{(x + \sqrt{a^2 - b^2})^2 + y^2}$$

Using the Equation 4.29 we can simplify this to $d = 2a$, which is a well-known property of the focal points. Interestingly, the sum of the distances from p to the conjugate complex focal points turns out to be $2b$. Since an arbitrary ellipse can be transformed into the standard form (4.29) by a rigid motion, this is true for the focal points of all ellipses. Similar results hold for the focal points of a hyperbola.

The definition of focal points as intersections of ideal tangent lines is quite interesting. For example, it shows that the Euclidean distance measure is not necessary to define focal points; the ideal points I and J suffice. In elementary Euclidean geometry, the focal points of a hyperbola are defined slightly different from those of an ellipse. Here, the definition does not depend on the type of conic. Concepts such as the major and minor axis, confocal conics etc become very easy to define.

Finally, we would like to point out that the definition of focal points generalizes to non-Euclidean geometries: the focal points of a conic C are still the intersections of the ideal lines tangent to C. However, when Ω is proper, the four tangents will intersect in *six* points. Thus, in non-Euclidean geometries, a conic has six focal points[3].

4.12.4 Affine geometry

In *affine geometry* we study the subgroup of the projective transformations which leaves a given line invariant. This invariant line[4] is called the *line at infinity*. In

[3] In Euclidean geometry the two "missing" intersection points are I and J, which are usually not regarded as focal points.

[4] The line is invariant as a line – it is not necessarily *point-wise* invariant.

contrast to Euclidean geometry, which also has a line at infinity, there is no metric in affine geometry.

All concepts that can be based on the line at infinity alone are part of affine geometry. For example, *parallel lines* are concurrent with the line at infinity (Section 4.11.3). The *midpoint* of a line segment pq is the harmonic conjugate of r with respect to p and q, where r is the intersection of the line pq and the line at infinity. Thus, both parallelism and midpoints of line segments are affine concepts and, by definition, they are left invariant by affine transformations. A number of theorems that are often associated with Euclidean geometry are actually affine, for example

- the diagonals of a parallelogram bisect each other, and
- the line joining the midpoints of two sides of a triangle is parallel to the remaining side.

However, the fact that the perpendicular bisectors of the sides of a triangle are concurrent is *not* an affine theorem since the concept of perpendicularity is not defined in affine geometry.

In contrast to projective geometry, there is no duality between points and lines since there is an invariant line but no invariant point. (Note that Euclidean geometry also lack duality while there is full duality in hyperbolic and elliptic geometries.)

Since there is a line at infinity associated with both affine and Euclidean geometry, we can think of Euclidean geometry as a special case of affine geometry where the absolute points I and J have been added and where the unit distance has been defined. However, there is no such relationship between affine geometry and hyperbolic or elliptic geometry. The infinity in hyperbolic and elliptic geometry is represented by a proper conic, and there are *two* points at infinity on every line. Therefore, affine theorems are in general not valid in non-Euclidean metric geometries.

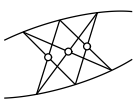
4.13 Orientation of projective elements

The elements of the real projective plane have no orientation. For example, a projective line has no forward direction. Consequently, it is not possible to speak about the line segment pq, where p and q are two points on a line.

Sometimes, however, it is convenient to assign a direction to each line and conic.

One way of orienting a line l is to select three distinct points p, q, r on l. The orientation of l is then the direction in which we can move from p to q without passing through r. This is of course closely related to the concept of separation (Section 4.9).

However, the concept of orientation can be given an algebraic definition which is more general and better suited for geometrical computations. The basic idea is



to keep track of the sign of the homogeneous coordinates. A line in the unoriented projective plane P_2 is identified with a one-dimensional linear subspace of R^3 and all vectors in that subspace represent the same line. We make no distinction between l and kl, where $l \in R^3$ and $k \in R,\ k \neq 0$. In the *oriented projective plane* T_2, however, l and kl are equivalent only if $k > 0$. Each line l in P_2 is replaced in T_2 by two distinct, oppositely oriented lines l and $-l$. $-l$ is called the *antipode* of l. Similarly, each point p in P_2 is replaced by two oppositely oriented points p and $-p$ in T_2. This results in a double covering of the projective plane, and T_2 is therefore also called the *two-sided projective plane*. The two sides are sometimes called the *front range* and *back range*. With the standard R^3 embedding of the projective plane (Section 4.1), a point

$$p = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$$

is said to be on the front range if $p_3 > 0$ and on the back range if $p_3 < 0$.

It turns out that it is possible to define a consistent oriented geometry, if we are careful with the signs [Stolfi91]. For example, the line defined by two points p and q (in that order) is

$$l = p \times q$$

which is the antipode of the line on q and p:

$$q \times p = -(p \times q) = -l$$

The orientation of points and lines makes it possible talk about the segment pq, the left and right side of a line, the positive turn at any point in the plane, convex sets etc. Similar definitions can be made in higher dimensions.

Oriented projective geometry has several applications. For example, when rendering a 3D scene in a computer graphics application, the use of an oriented projective space makes it possible to distinguish what is in front of from what is behind of the observer. In the next chapter, we will use oriented geometry to distinguish the intersection points of curves (Section 5.2.5) and to represent the motion of geometrical objects on the screen in a consistent way (Section 5.2.6).

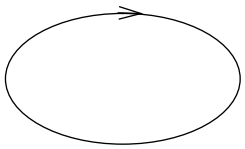


Figure 4.43. An oriented conic.

To that end, we will have to assign an orientation to each conic, depicted by the arrow in Figure 4.43. The theory as presented in [Stolfi91] deals only with linear subspaces (in the case of P_2, points and lines), not algebraic curves. Although a

conic can be considered as linear subspace of R^6 (Section 4.8.9), and therefore can be treated as an oriented point in T_5, that orientation is not directly related to the T_2 orientation depicted in Figure 4.43. Instead, we will define the orientation of a conic based on its embedding in R^3 (Section 4.8.8).

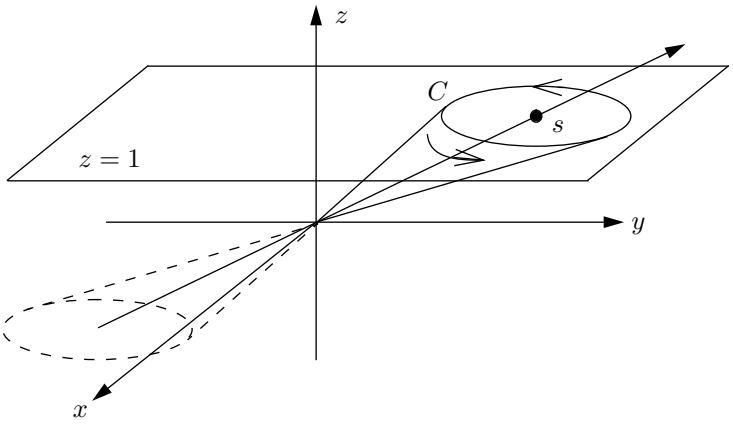


Figure 4.44. The orientation of the conic is defined using a directed symmetry axis.

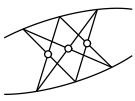
(a) s is on the front range.

(b) s is on the back range.

Figure 4.45. The orientation of the conic is determined by the orientation of s.

A real, proper conic C with a real point set (i.e., $C \in \mathrm{R}^{3\times 3}$ and $p^T C p = 0$ has real solutions) is represented in R^3 by a double, elliptic cone. The cone cuts out the conic in the plane $z = 1$, see Figure 4.44. If we assign a direction to the symmetry axis of the cone, we will at the same time define a direction of rotation of the cone: the rotation is given by a right-threaded screw moving in the direction of the axis. The direction of rotation in turn defines a direction along the curve of intersection in the plane $z = 1$. The symmetry axis in R^3 represents a projective point s. Making the axis directed is equivalent to making s oriented, i.e., treating s as an element of T_2. The direction along the conic is thus determined by the orientation of s, see Figure 4.45. For each conic with a given direction, there is another conic with the same point set rotating in the opposite direction.

How do we compute the symmetry axis of C? Assuming that C has real coefficients and a real point set, there is a matrix U such that $D = U^T C U$, D is



diagonal, and $U^TU = E$. Furthermore, if

$$D = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

U can be chosen so that d_1 and d_2 have the same sign and d_3 the opposite sign (Section 4.8.7). Thus, the equation $p^TDp = 0$ represents an ellipse centered around the origin. The symmetry axis of the corresponding double cone is obviously the z-axis. Furthermore, an orthogonal matrix preserves the inner product of R^3 (and Euclidean R^3 angles) since $(Up)^TUq = p^TU^TUq = p^Tq$. Thus, U rotates the double cone rigidly in R^3. Consequently, the symmetry axis of C must be the rotated symmetry axis of D, or

$$U \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Actually, the z-axis is the eigenvector of D which corresponds to d_3, the eigenvector that has a distinct sign. Since an orthogonal matrix preserves both eigenvalues and eigenvectors, the same holds for C. That is, C has three real eigenvalues. One of them, say λ has a distinct sign. The corresponding eigenvectors is the symmetry axis. (Since C is symmetric, the axis is orthogonal to the eigenvectors which correspond to the other eigenvalues.)

A Conic with no real points cannot be represented as a real double cone. For example, the imaginary unit circle is represented by

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

whose eigenvalues are all one. Hence, it is not possible to define a symmetry axis for that conic.

Finally, we note that the complex projective plane cannot be oriented. It is meaningless to define the antipode of p as $kp,\ k < 0$ since k can be complex.

4.14 The complex projective plane

One of pdb's main features is its ability to handle points, lines and conics with complex coordinates. In Section 5.3.4 we will discuss how such objects can be visualized on the screen. Basically, complex points and lines will be represented by points and lines in the real projective plane. We will try to find a mapping between complex and real coordinates which preserves as many incidence relationships as possible. It will therefore be important to know how geometric objects with real and complex coordinates interact. For example, how many real points does a complex line have and how many complex points does a real line have? A number of such facts will be given in this section.

4.14.1 Complex points and lines

We say that a point or a line in the complex projective plane $\mathrm{P}_2(\mathrm{C})$ is *real* if the one-dimensional subspace of C^3 it has been identified with is spanned by a real vector. That is, p is a real point if

$$\exists\, q \in \mathrm{R}^3,\ c \in \mathrm{C}\colon p = cq$$

Otherwise, we say that the point is *complex*. For example, $(1,2,3)^T$ and $(1+2i, 3+6i, -1-2i)^T$ are real points, while $(1,i,0)^T$ is complex.

It is easy to see that if $p = a + ib,\ a,b \in \mathrm{R}^3$, then p is real if and only if a and b are linearly dependent.

How many real points does a complex line have? Let the homogeneous coordinates of the line be $l = u + iv,\ u,v \in \mathrm{R}^3$, and let $p \in \mathrm{R}^3$ be a real point on that line. The line equation is then

$$p^T l = p^T(u + iv) = p^T u + ip^T v = 0$$

Since p, u and v are all real it follows that

$$\begin{cases} p^T u = 0 \\ p^T v = 0 \end{cases}$$

Since l was a complex line, u and v are linearly independent. Thus, we can consider u and v as the coordinates of two real, distinct lines. p must be on both of them, hence $p = u \times v$. This is the *only* real point on l.

We may also pose the opposite question: how many complex points does a real line have? A real line l has infinitely many real points. For each pair p, q of real points on l, the complex point $p + iq$ is also on l since

$$p^T l = 0,\ q^T l = 0 \Rightarrow (p + iq)^T l = p^T l + iq^T l = 0$$

Consequently, there is an infinite number of complex points on a real line.

Another important fact is that a complex point $p = a + ib$ and its complex conjugate $\overline{p} = a - ib$ defines a *real* line:

$$l = p \times \overline{p} = (a + ib) \times (a - ib) = a \times a + b \times b + i(b \times a - a \times b) = 2ib \times a$$

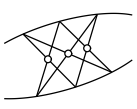
Since $2i$ is just a complex scale factor and a and b are both real, the line l is real.

Because of the duality of points and lines, the results above also apply to points: a complex point is on one and only one real line, a real point is on infinitely many complex lines, and the intersection point of two conjugate complex lines is real.

Can p and $\overline{p}$ represent the same projective point if p is complex, i.e., can we find a $c \in \mathrm{C}, c \neq 0\colon \overline{p} = cp$? Let $p = u + iv,\ u,v \in \mathrm{R}^{3\times 3}$ be a complex point and $c = a + ib,\ a,b \in \mathrm{R}$ be a scalar. Then

$$u - iv = \overline{p} = cp = (a + ib)(u + iv) = au - bv + i(av + bu) \Rightarrow u = au - bv$$

Since u and v are linearly independent $a = 1, b = 0$ and $\overline{p} = p$. That contradicts the assumption that p is complex. Thus, p and $\overline{p}$ represent the same projective point if and only if p is real.



4.14.2 Complex and real conics

In analogy with the definition of real and complex points, we say that a conic C is *real* if we can make its matrix real by multiplying it with a complex scalar. Otherwise, it is a *complex* conic. If $C = A + iB,\ A, B \in \mathrm{R}^{3\times 3}$, the conic which C represents is real if and only if A and B are linearly dependent.

It is important to note that with this definition, the point conic represented by the unit matrix E is real, although it contains no real points, i.e., the point equation $p^T C p = 0$ has no real solutions (Section 4.8.2).

A complex conic contains at most four real points. This follows from the fact that a conic is completely specified by five of its points. If there are five real points on the conic, we see from Equation 4.17 on page 44 that the conic is real. It can also be seen in the following way. If $p \in \mathrm{R}^3$, $C = A + iB,\ A, B \in \mathrm{R}^{3\times 3}$,

$$p^T C p = 0 \Leftrightarrow p^T (A + iB) p = 0 \Leftrightarrow p^T A p + i p^T B p = 0 \Leftrightarrow p^T A p = 0,\ p^T B p = 0$$

Thus, p is a point on C if and only if it is on the two real (possibly degenerated) conics A and B. Since two real conics can have no more than four real intersection points (Section 4.8.10), no more than four real points can be on C.

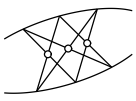
If C is real, how many complex points are on that conic? Since any complex line $l = u + iv$ intersects C in two (complex) points, there is infinitely many complex points on any conic C.

We showed in Section 4.8.3 that a conic and a line have two points in common. This analysis is valid also in $\mathrm{P}_2(\mathrm{C})$ (with $\lambda \in \mathrm{C}$). If a real conic C and a real line l have a point p in common, they must also have $\overline{p}$ in common since

$$p^T C p = 0,\ \overline{C} = C \Rightarrow \overline{p}^T C \overline{p} = 0$$
$$p^T l = 0,\ \overline{l} = l \Rightarrow \overline{p}^T l = 0$$

Since there are only two intersection points, a real conic and a real line intersect in either two real points or in two conjugate complex points. However, a complex line will not intersect a conic in two conjugate complex points since such points define a real line (i.e., $p \times \overline{p}$ is a real line).

Two complex conics intersect in four points. If p is one intersection point and the conics are real, $\overline{p}$ is also an intersection point. Thus, the intersection points of two real conics are either all real or pair-wise conjugate complex.



Chapter 5

Designing a dynamic geometry system

5.1 Handling constraints

Drawings consisting of points, lines and algebraic curves can be created by many graphics systems, ranging from word processors to advanced CAD systems. What is specific to a *dynamic geometry system* is that it allows the user to place constraints on the positions of objects. The user may require, for example, that a certain line must be tangent to a certain conic, and the system will make sure that this constraint is always satisfied.

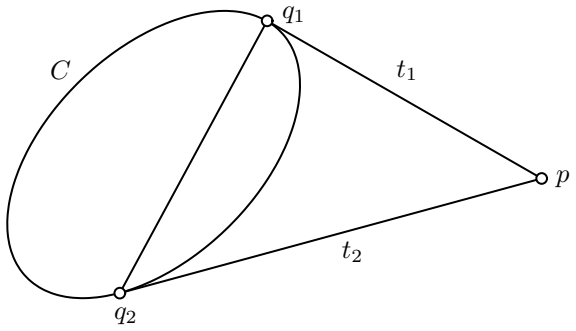
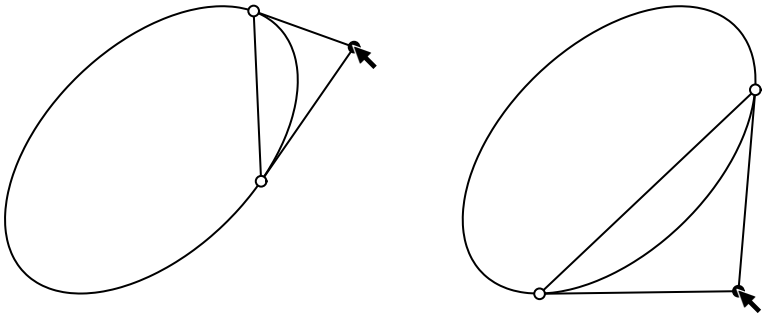


Figure 5.1. The polar of a point with respect to a conic.

Consider the drawing in Figure 5.1, where the *polar* of a given point p with respect to a given conic C has been constructed. First, the two tangents t_1 and t_2 intersecting each other at p were drawn. Then, the two tangent points q_1, q_2 were connected by a third line, which is the polar of p. If the drawing in Figure 5.1 had been a dynamic sketch, we could have dragged the point p and watched the polar line move, as indicated by Figures 5.2a-b.



(a) *(b)*

Figure 5.2. Dragging the pole.

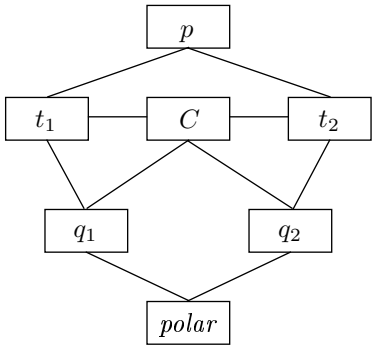
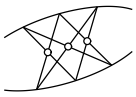


Figure 5.3. A constraint graph representing a polar line construction.

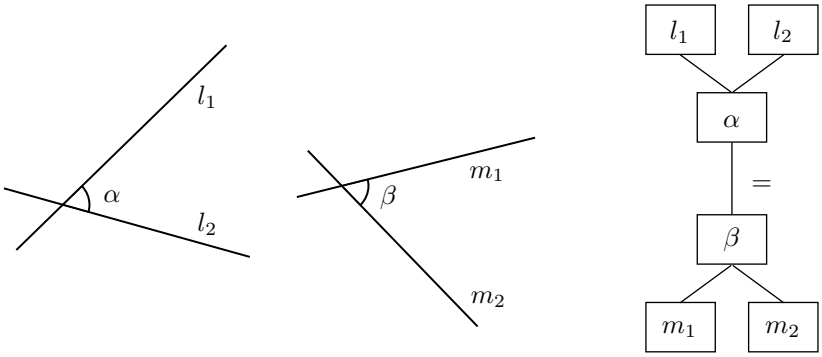
How can such a dynamic sketch be implemented? What kind of representations and algorithm are required? In Section 5.1.1, we will look at the possibility of using *constraint programming* techniques. A number of problems with this approach, in particular those related to under-constrained systems, will be discussed in Section 5.1.2. In Section 5.1.3, we will suggest an alternative approach.

5.1.1 Dynamic geometry as a constraint programming problem

In a constraint programming system, the point, the lines and the conic in Figure 5.1 would be represented by nodes in a *constraint graph*. The constraints would be represented by arcs between the nodes as shown in Figure 5.3. In this case, the arcs represent tangency and collinearity constraints. Connected nodes cooperate to determine the positions of the corresponding geometrical objects, so that all constraints are satisfied. If an object is moved, the corresponding node will notify its neighbors and new coordinates will be computed for all affected objects. The constraint graph is not directed; information can flow in any direction



and an updating sequence can be initiated by any node. For example, if the user would pick and drag the polar line in Figure 5.2, the position of the pole would be changed. This bi-directional propagation of information makes it easy to work with simple sketches such as the pole-polar sketch. As we shall see, however, an unrestricted propagation of information can be a problem in more complicated sketches, since it gives rise to ambiguities.



(a) Geometric view. *(b) Constraint graph.*

Figure 5.4. An angle equality constraint.

Measurements, such as distances, angles and cross-ratios can also be represented by nodes in the constraint graph. In Figure 5.4a, we have drawn two pairs of points. The angles between the lines in each pair, α and β, have been restricted by an equality constraint. The corresponding constraint graph, shown in Figure 5.4b, is very simple. If a line in one of the line pairs is dragged, the position of at least one of the lines in the other pair will be updated, so that the angles remain equal.

How can a constraint programming system be implemented? From an algebraic point of view, the nodes of the constraint graph are variables, and the arcs are equations that these variables have to satisfy. Each connected component in the constraint graph represents a system of equations that has to be solved whenever a node (i.e., a variable) is updated. Incidence constraints between points and lines can be represented by linear equations. However, constraints involving conics, angles and cross-ratios give rise to non-linear equations. Therefore, we are in general faced with a large system of non-linear polynomial equations in several variables.

For example, consider again the polar line construction in Figure 5.1. To simplify the problem slightly, let us consider the positions of the given point p and the conic C fixed. That reduces the number of free variables and thereby the number of constraints required to obtain a unique solution. We have the following

constraints: The lines t_1 and t_2 must intersect in p:

$$p^T t_1 = 0 \tag{5.1}$$
$$p^T t_2 = 0 \tag{5.2}$$

The lines t_1 and t_2 must be tangents to C:

$$t_1^T C^{-T} t_1 = 0 \tag{5.3}$$
$$t_2^T C^{-T} t_2 = 0 \tag{5.4}$$

The tangent points q_1 and q_2 depend on both the conic and the lines t_1 and t_2:

$$q_1^T t_1 = 0 \tag{5.5}$$
$$q_2^T t_2 = 0 \tag{5.6}$$
$$q_1^T C q_1 = 0 \tag{5.7}$$
$$q_2^T C q_2 = 0 \tag{5.8}$$

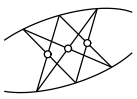
Finally, the polar line l must be incident with both tangent points:

$$q_1^T l = 0 \tag{5.9}$$
$$q_2^T l = 0 \tag{5.10}$$

A non-linear system of polynomial equations in several variables can be solved either numerically or by algebraic methods. To solve the system algebraicly, one can rewrite the equations in a different basis, G, so that the system is converted to a triangular form. It is then relatively easy to solve for one variable at the time, using back-substitution. G is called a *Gröbner basis* for the system and can always be found using the *Buchberger algorithm* [Cox92, Hägglöf95]. Solving non-linear systems using Gröbner bases is somewhat similar to using Gaussian elimination and back-substitution for solving linear systems.

The advantage of using Gröbner bases is that we get to know everything there is to know about the given system of equations. In particular, we can see how many solutions that the system has, and we can compute all of them. A problem is that determining the Gröbner basis in the general case is very costly. In fact, the *Buchberger algorithm* is NP-complete.

To solve the system of equations numerically, one can use a standard iterative method such as Newton-Raphson [Dahlquist74]. However, an iterative method requires an initial guess, which should be sufficiently close to the desired solution. Some systems let the user provide the initial guess by letting him draw a sketch where the constraints are approximately satisfied. Then the system adjusts the drawing so that all constraints become completely satisfied. Numerical methods and symbolic manipulations can also be used in combination to speed up the computations [Heydon94].



5.1.2 Interacting with under-constrained drawings

A system of equations is said to be under-constrained if it has more than one solution. The number of solutions can be finite or infinite. For example, the system of equations in Section 5.1.1 contains no constraint that prevents lines t_1 and t_2 from coinciding. Therefore, there are two solutions where $t_1 = t_2$ and $q_1 = q_2$, see Figure 5.5a and b. In those cases, the orientation of the polar line l is undetermined. Thus, one additional constraint is required to make the solution unique: the inequality $t_1 \neq t_2$. If a system of non-linear equations is solved by numerical methods, it is difficult to detect that the system is under-constrained. Most numerical methods converge to a single solution, even if there are infinite many. The solution found will depend on the initial seeds. With algebraic methods, however, it is possible to determine whether a unique solution exists or not. If Gröbner bases are used, a complete parameterization of the solution space can be obtained.

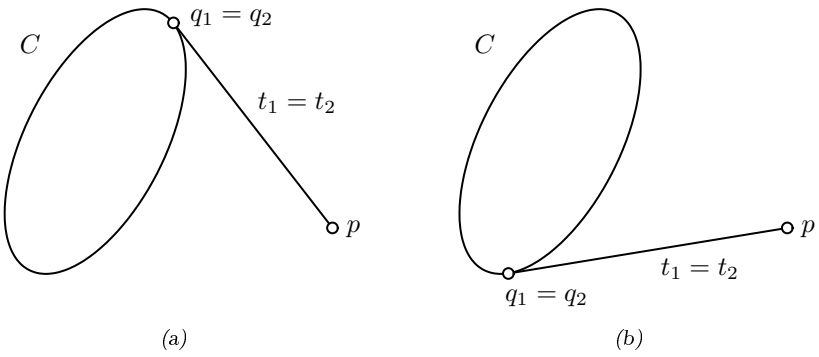


Figure 5.5. A collapsed polar line constructions.

If a dynamic geometry system discovers that a part of a drawing is not fully specified, it may enter a dialog with the user and ask for additional constraints, or it may ask the user to choose one of the possible configurations. This behavior has been proposed for solving ambiguity problems in UniGéom (Section 2.5). For geometry systems intended for constraint-based CAD, that may be appropriate. CAD users are typically looking for a unique solution, an object with well-defined properties such as minimum weight and size, a certain smoothness etc. The user will continue to add constraints until the system confirms that only one solution remains. In contrast, sketches drawn in a dynamic geometry system are almost always under-constrained. This is because the whole idea of a dynamic geometry system is to allow the user to drag objects around and watch the dynamics of the construction. That would not be possible if the position of all objects were fully specified by the constraints; at least some of the objects have to have a degree of freedom.

Thus, under-constrained drawings are fundamental to dynamic geometry. Given a set of user-defined constraints, the program will display *one* of the ge-

ometric configurations that satisfy the constraints. By dragging objects on the screen, the user should be able to move smoothly through a continuum of allowable configurations. For example, if the user picks a line by pressing the mouse button while the cursor is over the line, the system should make sure that the line remains incident with the cursor as long as the button is held down. When the cursor is moved, the line will follow. By picking the line, the user has provided the program with a new, temporary incidence constraint which forces the program to select a different geometric configuration each time the cursor is moved. The constraint is removed as soon as the user lets go of the line by releasing the mouse button.

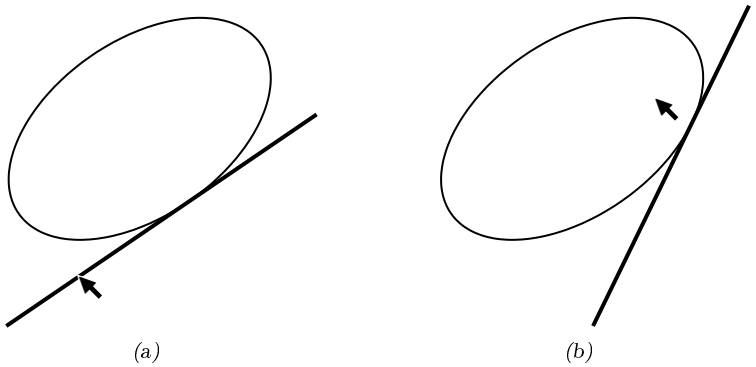
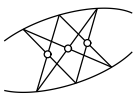


Figure 5.6. If the tangent of an ellipse is dragged and the cursor is moved inside the conic, the tangent cannot stay incident with the cursor.

However, an extra incidence constraint will not always be consistent with the existing constraints. For example, suppose the user picks a line that is required to be tangent to an ellipse, and then moves the cursor inside the ellipse, see Figure 5.6. Obviously, the line cannot be tangent to the ellipse and incident with the cursor at the same time[1]. In this case, it might be reasonable to require that the line passes through the point on the ellipse that is closest to the cursor position. Thus, the constraint added during dragging operations must be chosen carefully, in order to avoid inconsistencies. All existing constraints on the object being dragged must be taken into account.

On the other hand, the extra constraint implied by the dragging operation will not always be enough to produce a unique solution. If the resulting system of equations remains under-determined, the program still needs to fixate the position of one or several objects each time the drawing is updated. In practice, the program will do that by adding temporary constraints until only one solution remains. These constraints will be chosen by the program, not by the user, and will be removed automatically as soon as new positions have been computed for all objects. The choice of constraints is somewhat arbitrary, but should be based

[1] Unless it is complex. However, the screen image of the line will always be real.



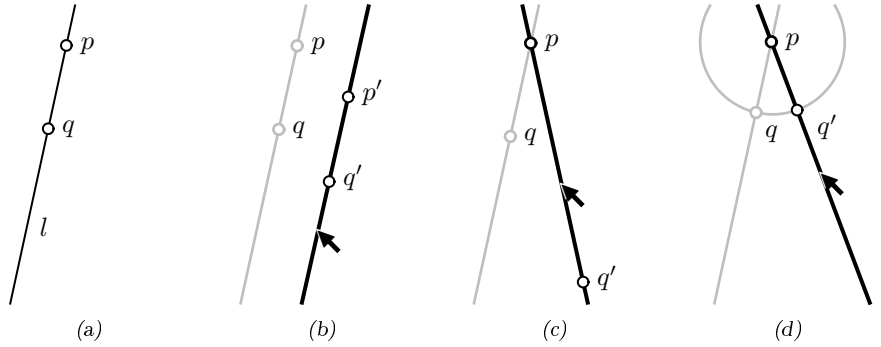


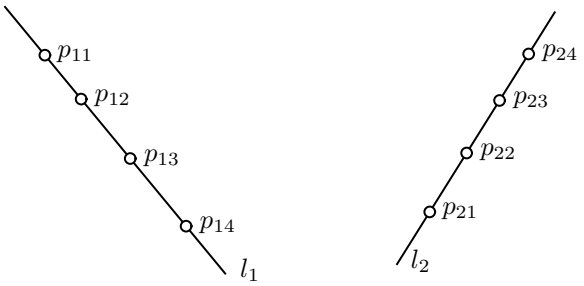
Figure 5.7. Dragging a line with two points.

on two principles:

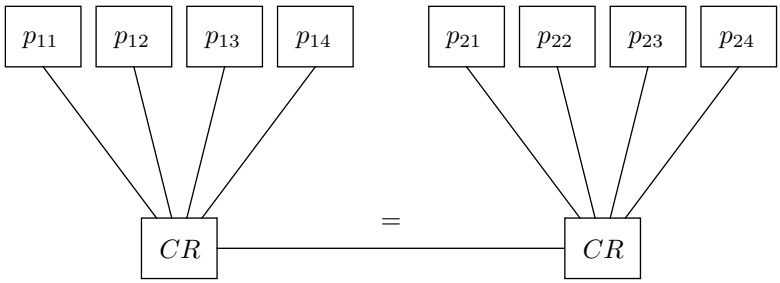
- When an object is dragged, the motion of other, dependent objects should be predictable and intuitive to the user. In particular, the position of the objects should be a continuous function of the cursor position. It is also desirable that all objects remain visible on the screen, if possible.
- The user should be able to reach every configuration that is consistent with the explicitly defined constraints through a finite sequence of dragging operations. It should also be obvious to him how he should drag the objects in order to achieve a certain configuration.

To illustrate this, let us look at an example. Consider the simple sketch in Figure 5.7a which consists of two points p and q and a line l. Suppose that the line l has been restricted to be incident with p and q, or equivalently, p and q have been restricted to be incident with l. There is only one constraint equation, $l = p \times q$, so the system is clearly under-constrained. Suppose the line l is dragged into a new position. What does the user expect will happen to the other objects? We cannot be sure, but the effect shown in Figure 5.7b, where the points have been translated by the same distance as the line, seems natural to most users. Suppose now that the position of the point p is fixated by an explicit constraint, and that q and l remain free. It is then clear that the line l must rotate around p if it is dragged, but what should happen to the point q? The system is free to place q anywhere on the line l and the image of q might very well slide rapidly along the line and out of view (see Figure 5.7c). This effect can be observed in many dynamic geometry systems. It not only annoying, but it actually makes it very difficult to interact with the drawing. Here, it is reasonable to require that the system keeps the distance between p and q constant (Figure 5.7d). Both in Figure 5.7b and in Figure 5.7d, the user gets the impression that the configuration is rigid, which makes the motion of the under-determined objects more predictable. However, the user must still be able to change the distance between the points; if the user picks and drags q, the system might choose to keep p and l still. In that case, the configuration will not and should not appear to be rigid.

Thus, in order to resolve ambiguities, the program has to add and remove extra constraints on the fly, as the user picks and drops objects on the screen. The choice of these constraints will depend on the type of objects that the user is interacting with and the constraints previously defined by the user. To make this possible, the constraints must be represented internally so that the geometric interpretation of the equations are made *explicit*. It would be an extremely difficult task for a program to analyze a large, under-constrained, non-linear system of equations and based on that analysis select additional constraints that will produce not only a unique solution, but the solution expected by the user. The polar construction example above (Figure 5.5) indicated how difficult it can be to find the right set of constraints at the algebraic level. In fact, one of the main problems with using the Gröbner bases (Section 5.1.1) in this context is that when the given system of equations has been transformed into triangular form, the equations can no longer be given a simple geometric interpretation. Thus, when using Gröbner bases, we can easily see that a system is under-constrained, but it will not be obvious how we should make it well-constrained.



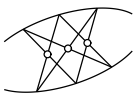
(a) Geometric view.



(b) Constraint graph.

Figure 5.8. Two sets of collinear points with equal cross-ratio.

Sometimes there is so little information available that the program cannot guess what the user is trying to accomplish by a dragging operation. Figure 5.8a shows two lines l_1 and l_2, with four points on each line (p_{11} through p_{14} and p_{21} through p_{24}). Assume that the points p_{ij} are restricted to be incident with the



line l_i, and that the cross-ratios $(p_{11}\ p_{12} \mid p_{13}\ p_{14})$ and $(p_{21}\ p_{22} \mid p_{23}\ p_{24})$ must be equal. The constraint graph is shown in Figure 5.8b and the corresponding system of equations is

$$\begin{cases} p_{ij}^T l_i = 0 & i = 1, 2,\ j = 1, 2, 3, 4 \\ (p_{11}\ p_{12} \mid p_{13}\ p_{14}) = (p_{21}\ p_{22} \mid p_{23}\ p_{24}) \end{cases}$$

Now, assume that the user picks point p_{11} and drags it. How should the drawing be updated? In this case, the system might assume that the user does not want to change the position of the lines. If the program chooses to keep the remaining points on l_1 fixed, the cross-ratio will be changed. Then the position of at least one point on l_2 must be modified, but which one? On the other hand, it may be just as reasonable to keep the cross-ratio fixed when p_{11} is dragged, and instead update the position of the remaining points on l_1, i.e., p_{12}, p_{13} and p_{14}. This ambiguity cannot be resolved automatically. The user must tell the program what he intends to do by manually placing extra constraints on the objects. For example, he could tell the system that the position of the points p_{12}, p_{13} and p_{14} should be fixated at their current positions. Of course, if he later decides to drag one of those points, he would first have to remove those extra constraints again.

Adding and removing constraints manually to fixate the positions of certain objects during updates is very cumbersome. One way to avoid it is to restrict the flow of information in the constraint graph. When creating the drawing in Figure 5.8a, we might want to specify, once and for all, that all points on l_1 should be movable, that we only want to *measure* their cross-ratio, and that this cross-ratio should *determine* the position of, say, p_{24} relative to p_{21}, p_{22} and p_{23} so that $(p_{21}\ p_{22} \mid p_{23}\ p_{24}) = (p_{11}\ p_{12} \mid p_{13}\ p_{14})$. This gives the system a lot of extra information: the position of p_{24} is completely determined by the other points. All points except p_{24} are freely movable along the lines, and if one of them is dragged, only p_{24} should be affected. This completely specifies what should happen during dragging. No other constraints will be necessary.

5.1.3 A construction-oriented approach

We made three important observations in the previous sections:

- Algebraic methods for solving large systems of non-linear equations are slow and numerical methods are difficult to use for under-constrained systems.
- When updating under-constrained drawings, the program needs to add extra constraints automatically in order to produce a unique solution. That will be feasible only if the geometric interpretation of the constraint equations has been made available to the program.
- Unrestricted propagation of information in the constraint graph may create ambiguities that are difficult for the program to handle.

We have therefore adopted a different strategy in pdb. Instead of having a general constraint graph where information can flow in any direction, we have chosen a simpler type of *directed acyclic graph* (DAG) where information only flows in one direction. Furthermore, each node in this graph is a *combination* of a graphical object (e.g., point, line, conic, angle) and one or several constraints (e.g., incidences, tangencies, equalities).

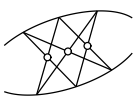
Figure 5.9 shows the construction of a simple drawing using pdb. First, two points p and q are placed on the drawing board (Figure 5.9a). The positions of the points are not constrained, so the user can move them freely over the screen. The corresponding "constraint graph" is shown in Figure 5.9b. It consists of two separate nodes of a type called `FreePoint`. In Figure 5.9c, a line l has been attached to the points. The constraint graph now consists of three nodes, as shown in Figure 5.9d. A new node of type `LineOnTwoPoints` has been connected to the point nodes. This means that the line l will always be incident with the the two points; if one of the points is moved, the line will follow. The line node will simply fetch the current position of the points and update its own position accordingly, using the formula $l = p \times q$. Thus, information flows in the direction of the arrows in Figure 5.9d; never in the reverse direction. Consequently, the user will not be able to change the position of the points by dragging the line. We will call the points p and q the *parents* of the line l, and l the *child* of p and q. Since the graph basically shows how the objects depend on each other, we will call it a *dependency graph*.

Next, a point r is attached to the line l, as shown in Figures 5.9e-f. A new node, called `PointOnOneLine` has now been connected to the line node. Whenever the position of the line is changed (because one of the points p and q is dragged), the point r will update its position so that it stays on its parent line l (Figure 5.9g). However, r still has a degree of freedom – it can be dragged along l (Figure 5.9h).

Thus, we have shapes that are completely unconstrained (the `FreePoint`), shapes whose positions are completely determined by the position of their parents (the `LineOnTwoPoints`), and shapes whose position is only partly determined by their parents (the `PointOnOneLine`).

The position of r on the line l is not completely determined by the constraints placed on r. However, the `PointOnOneLine` node understands the geometric situation and can add another, temporary constraint when the position of r needs to be updated. These extra constraints will be discussed in Section 5.2.6, and we will see that, for example, it is reasonable to preserve the relative distances between r and the points p and q when l moves. The `PointOnOneLine` node will use this constraint automatically, without any user intervention.

Now, return to the polar line of Figure 5.1. The representation of this construction in pdb is shown in Figure 5.10. The top nodes represent the given point p (here, a `FreePoint`) and the given conic C (here, a `FreeConic`). The two tangent lines are represented by two `LineOnConicAndPoint` nodes, which are children of p and C. The tangent points are represented by two `PointOnConicAndLine` nodes, which are children of the conic and the tangent lines. Finally, the polar line is represented by a `LineOnTwoPoints` node, which is the common child of the



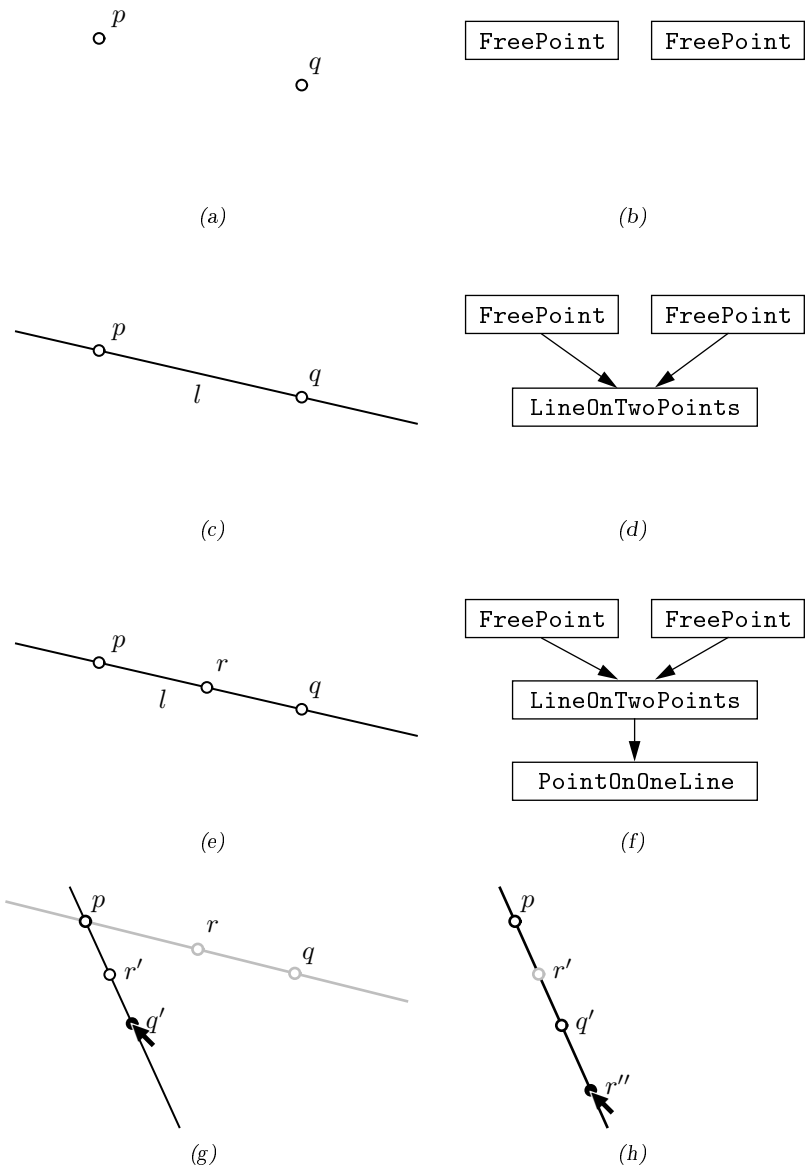


Figure 5.9. A simple drawing implemented using a construction-oriented approach.

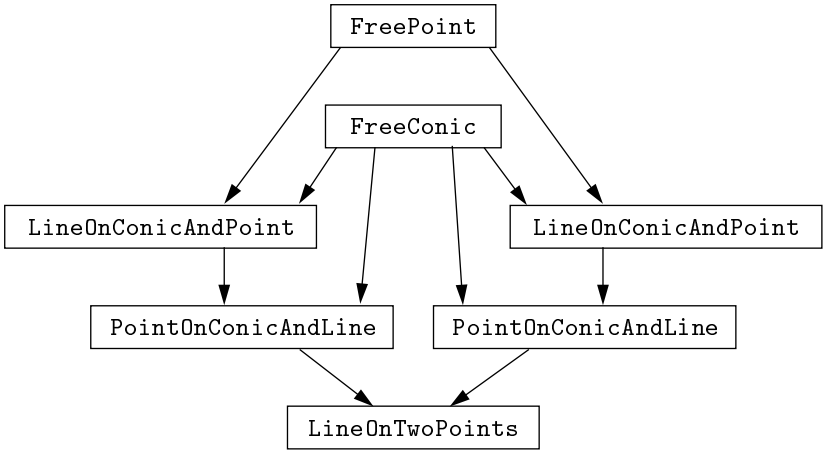


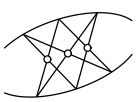
Figure 5.10. The representation of the polar line construction in pdb.

two tangent points. When p is dragged, the position of all dependent nodes are updated in turn. In this graph, only the conic and the point p have some degree of freedom. The position of all other shapes are completely determined by their parents.

5.1.4 Comparing pdb with constraint-based systems

When creating a dynamic drawing, a user of a system based on general constraint programming would first create a set of shapes (points, lines, conics), then add suitable constraints to fixate their positions. The drawing would be represented as a set of variables (object positions) and a set of equations (constraints). In contrast, the directed, acyclic dependency graph used in pdb represents the "constructional history" of the drawing, one of the main ideas in [Naeve89]. The nodes, each one being a combination of a shape and a constraint, are added to the graph in the same order in which the user creates the corresponding objects on the drawing board surface. The resulting graph will closely match a geometer's mental image of a ruler-and-compass drawing. One could say that users of constraint-based systems are encouraged to think of drawings in terms of specifications while pdb encourages its users to think in terms of geometric constructions.

Constraint-programming systems have some advantages. Thanks to the free propagation of information in the constraint graph, the user interface can be made more flexible. In pdb, the only way to move line l in Figure 5.9 is to pick and drag one of the points p or q. In a constraint programming system, it would be possible to drag l and let the system update the position of p and q. Also, it is not always obvious how to construct a drawing. For example, given three concurrent lines l_1, l_2 and l_3, how do we *construct* the triangle for which the lines are the perpendicular bisectors of its sides? The drawing is much easier to *specify* as a set of constraints on the vertices a, b and c of the triangle (see Figure 5.11), such



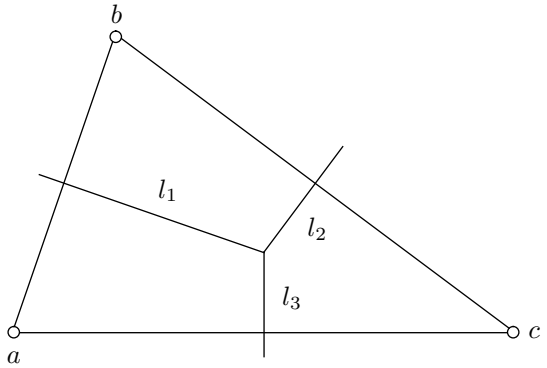


Figure 5.11. A triangle determined by the perpendicular bisectors of its sides.

as

$$\begin{cases} a \times b \perp l_1 \\ b \times c \perp l_2 \\ c \times a \perp l_3 \end{cases}$$

On the other hand, there is usually an obvious way of constructing a drawing step-by-step, and in that case, it is just extra work to formulate a set of constraints that define the drawing. Actually, it can be quite difficult to come up with a *consistent* set of constraints. The creators of the constraint-based drawing editor Juno-2 (Section 2.6) have noted in [Heydon94] that "the effective definition of constraints can require more mathematical sophistication than most users have".

The main advantage of using a DAG representation is that the geometric interpretation of the constraints is made explicit. The ambiguities in the specification of the drawing are not a hidden property of a large system of equations but represented by free parameters in certain object types, such as `PointOnOneLine`. When updating the drawing, the system can use this information to define extra, temporary constraints which uniquely determine the new position of every object and at the same time make the motion of objects predictable and intuitive to the user. It also makes it possible for the system to provide effective feed-back when the user wants to add or remove constraints.

Since the DAG representation of the drawing is constructive, the computations can be made very fast. Because every node in the directed graph is independent of its descendants, it is not be necessary to solve a large system of equations each time a node is added. The new node can express the position of the geometric primitive it represents in terms of the coordinates of its parents, usually in closed form. The order in which the nodes of the graph are updated can also be precomputed; when the information in a node is changed, its descendants will be requested to update their internal information only once.

The main problem with the combined shape and constraint representation in pdb is that the number of node types required can be very large. In principle, a

new node type will be required for every combination of object (point, line, conic, angle, distance) and constraint. However, we can get away with a fairly limited number of primitives (Section 5.2.1), and rely on macros (Section 5.3.6) for more complicated constructions.

5.2 Mathematical model and internal representation

5.2.1 Shape primitives and constraints

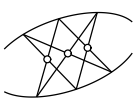
How many types of nodes are needed for creating dynamic drawings involving points, lines and conics? Let us for a moment ignore the conics and just concentrate on the points and lines. Clearly, we need the node types `FreePoint`, `PointOnOneLine` and `PointOnTwoLines`. The duals will then be called `FreeLine`, `LineOnOnePoint` and `LineOnTwoPoints`, respectively. Actually, it would suffice to implement only the last type of line, `LineOnTwoPoints` since two extra `FreePoint` instances and a `LineOnTwoPoints` child could serve as a "free line". However, if we want the system to be able to automatically dualize a drawing involving free points and points on one line, we must implement node types representing all duals.

Conics can be defined in many more ways. What we call a `FreeConic` is a conic that is defined by a coefficient matrix. It has no parents. A conic can also be defined by five of its points (`ConicOnFivePoints`), by four of its points and one of its tangent lines (`ConicOnFourPointAndOneLine`) or by three of its points and two of its tangent lines (`ConicOnThreePointsAndTwoLines`). However, while a conic is uniquely determined by five points, there are two conics that go through four points and touch one line, and four conics that go through three points and touch two lines. Thus, a `ConicOnFourPointAndOneLine` and a `ConicOnThreePointsAndTwoLines` have one (discrete) degree of freedom. The line duals of these conics are called `ConicOnFiveLines`, `ConicOnFourLinesAndOnePoint`, and `ConicOnThreeLinesAndTwoPoints`, respectively[2]

We have not yet attempted to implement conic/conic tangencies in pdb, although such constraints would certainly be very useful. For example, circles in hyperbolic geometry are conics which have double contact with the absolute conic (Section 4.11).

A conic can be a parent of a point or a line. As we saw in Section 5.1.3, the points where a line intersects a conic are represented by the `PointOnConicAndLine` nodes. Since there are two possible solutions, a `PointOnConicAndLine` has one discrete degree of freedom. Two conics define four intersection points in general. These are represented by `PointOnTwoConics` nodes. As a convenience, we have also implemented `PointOnOneConic`, which has one continuous degree

[2] In the current prototype, pdb version 2.2, only `FreeConic`, `ConicOnFivePoints` and `ConicOnFiveLines` are operational.



of freedom (it can be dragged along the conic). The line duals are called `LineOnConicAndPoint`, `LineOnTwoConics` and `LineOnOneConic`, respectively.

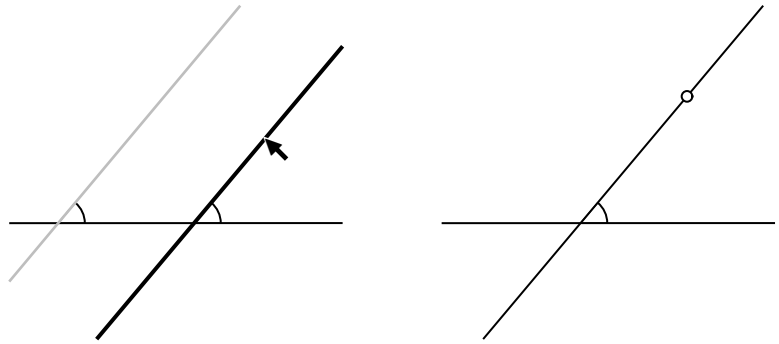


Figure 5.12. A `LineWithAngle` *can be translated but not rotated (a).*
A `LineOnPointWithAngle` *has no degree of freedom and cannot be dragged (b).*

So far we have defined 19 node types, which only deals with incidence relationships. A few more nodes are needed for representing metric constraints. Again, concentrating only on points and lines, we want to be able to restrict a free line to have a certain angle with respect to another line. Thus, we need a `LineWithAngle` node. The parent of a `LineWithAngle` serves as a base line, from which the angle is measured. The `LineWithAngle` has one degree of freedom; it can be translated, but not rotated, see Figure 5.12a. Sometimes, we also want the line to pass through a certain point. That adds another constraint, and we need to define a `LineOnPointWithAngle` node type, which has no degree of freedom (Figure 5.12b). The point duals are called `PointAtDistance` and `PointOnLineAtDistance`, respectively.

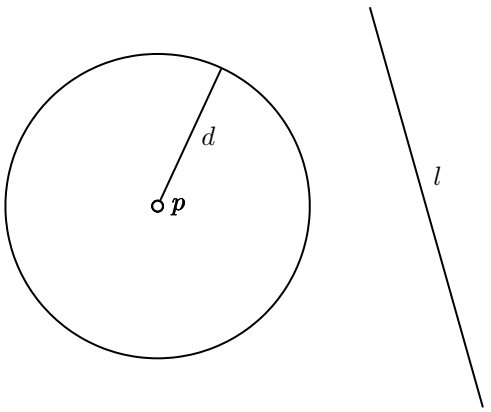
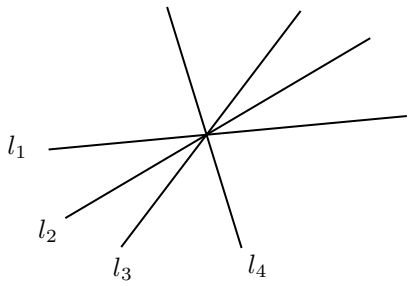
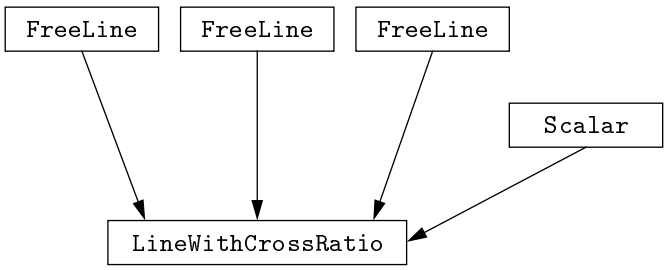


Figure 5.13. There might be no point on l at distance d from p.

`PointOnLineAtDistance` and `LineOnPointWithAngle` introduce a slight mathematical complication. Given a point p, a line l and a distance d, there is not always a point q on l such that distance to p is d, see Figure 5.13. Thus, `PointOnLineAtDistance` is one of the few node types that must be prepared to handle cases where there are no solutions to the constraints they represents. `LineOnPointWithAngle` does not suffer from that problem provided that the angle metric is Euclidean. Given a line l, a point p and an angle a, there is always a line m on p whose angle to l is a. This is because the set of lines cutting a fixed line at a constant (non-zero) angle will sweep the entire projective plane. However, that is not true in general. In hyperbolic geometry, for example, the envelope of a set of constant-angle lines is a conic which has double contact with the absolute. For straight angles and right angles the conic is degenerate. Obviously, points inside the conic are not on any line in the set.



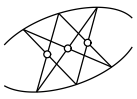
(a) Geometric view.



(b) Internal representation.

Figure 5.14. Given three intersecting lines l_1, l_2, l_3, a scalar value can be interpreted as the coordinate of a fourth line l_4.

The cross-ratio of four lines that intersects in the same point was defined in Section 4.3. Given three lines l_1, l_2, l_3 and a cross-ratio r, we can determine l_4 so that $(l_1\, l_2 \,|\, l_3\, l_4) = r$. Therefore, we have introduced `LineWithCrossRatio`, which has four parents: three other lines (which must intersect in the same point), and a value representing the cross-ratio, see Figure 5.14a and b. Whenever one of the three parents lines or the scalar value is changed, the position of the fourth line



will be updated. The point dual is called `PointWithCrossRatio`.

Could we manage with a smaller set of node types? In principle, yes. For example, the tangent to a conic through a given point can be constructed from `PointOnConic`, `PointOnConicAndLine`, and `LineOnTwoPoints` as shown in Figure 5.15. First, we pick three points r_1, r_2, and r_3 on the conic. Then the lines l_i through p and r_i are drawn. Let the second intersection point of l_i and the conic be s_i. Let u_1 be the intersection of segments r_1s_2 and r_2s_1, and u_2 be the intersection of r_2s_3 and r_3s_2. Finally, draw the line m through points u_1 and u_2. If m intersects the conic in q_1 and q_2, then pq_1 and pq_2 are the two tangents through p. In fact, some tools, such as Cabri (Section 2.1), actually implement the construction as a macro. However, the construction will collapse if, for example, r_1 and r_2 come too close together. The construction is not numerically stable and the auxiliary points and lines must be carefully selected. Therefore, we have chosen to implement the `LineOnConicAndPoint` as a primitive, whose coordinates can always be computed using a numerically stable algorithm. Also, drawings involving a large number of tangents will be updated significantly faster when `LineOnConicAndPoint` is a primitive. (The tangent construction in Figure 5.15 involves no less than 18 points and lines, all of which must be updated in each cycle.)

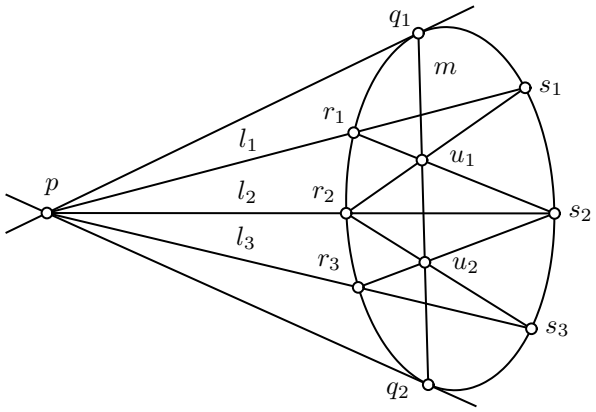


Figure 5.15. The construction of the tangents of a conic through a given point p.

There is also another reason for making `LineOnConicAndPoint` a primitive. As we mentioned earlier, there are actually *two* lines that go through a given point p and touch a given conic C, and it is important to differentiate between the two. If the tangent line is constructed by a macro, the result will depend subtly on how the macro is written, and the user will not be able to drag the tangent line between its two alternative positions.

Nodes for angle and distance constraints are not strictly necessary since it is possible to put constraints on cross-ratios in pdb. Instead of directly specifying the angle between two lines which intersect in a point p, we could specify the cross-ratio of four lines: the two given lines and the two ideal lines on p. However,

distance constraints can not be handled that easily in Euclidean geometry, where there is no complete duality between points and lines and between distances and angles. Also, angles and distances are easier to represent graphically on the screen, and constraints involving angles and distances can be defined using a drag-and-drop interface.

In some cases we have used macros to implement constructions that are primitives in other systems. For example, the construction of a polar line is a primitive in Cinderella Café (Section 2.3). Given a point p and a conic C, the coordinates of the polar line are $l = Cp$. Because of the simple form and the good numerical properties of this equation, it is quite natural to make it a primitive. We have chosen to implement it as a macro, though, because it can be constructed from `LineOnConicAndPoint`, `PointOnConicAndLine` and `LineOnTwoPoints` nodes, without placing auxiliary points and lines at arbitrary positions. Also, there is only on polar line for each given point, so there is no ambiguity that the macro has to deal with.

5.2.2 Supporting complex coordinates

As mentioned in Section 3.1, pdb can handle points, lines and conics with complex coordinates. All computations are done in complex arithmetic, and all algorithms work for complex coordinates. Geometric constructions in the complex projective plane can therefore be supported.

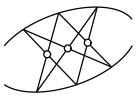
However, objects with complex coordinates are much harder to interact with. A complex point has four real dimensions, which is more than can be visualized faithfully on a flat computer screen. Furthermore, a pointing device such as a mouse provides input in just two real dimensions.

Because of these difficulties, the user interface of pdb was designed basically for real projective geometry. The use of complex arithmetic is primarily intended to simplify the treatment of real geometric constructions. Special cases can be avoided since the system can handle complex solutions of equations. An example is the polar line in Figure 5.1, page 81, where the same geometric construction works both for interior and exterior points thanks to the complex arithmetic. The support for complex coordinates also makes it possible to use absolute elements such as the complex circular points I and J in Euclidean geometry. That allows us, for example, to construct a set of confocal conics using only incidence constraints (Section 4.12.3).

The support for displaying and interacting with complex objects is limited. In situations where objects with complex coordinates must be displayed, they are mapped to the real plane and shown on the screen in a different color. The mapping can be defined in many ways, which will be discussed in Section 5.3.4.

5.2.3 Common interaction problems

When using dynamic geometry systems, you will probably see strange things happen from time to time. An object might suddenly jump into a different position



or disappear completely. A group of objects may converge to the same position – drawn to each other by invisible forces, they end up in a pile on the screen. At first you might think that this is caused by some easy-to-fix bugs in the program. However, it turns out that a number of non-trivial mathematical problems have to be addressed in order to work out a satisfactory solution. It is no coincidence that these defects are present in virtually every existing dynamic geometry system.

In this section we will look at a number of problems that can occur when a user is dragging objects around on the screen. In the following sections we will suggest how to avoid them. The problems we will talk about here have one thing in common: they are caused by the presence of under-constrained objects such as movable points on a line, movable tangents of a conic, etc. There are no problems with objects whose positions are completely determined by constraints.

Drifting objects

If the user picks a line and starts dragging it, points that have been attached to it will also be moved so that they stay on the line. It is not uncommon in geometry systems that under-constrained points on a line fail to return to their original positions if the line is returned to its original position. As a result, under-constrained points will drift along the line if the line is jogged. Usually, the points come closer and closer together, but they can also drift apart. Consider the sketch of the theorem of Pappus in Figure 5.16a. After rotating the top line a few laps, we may very well end up with the configuration shown in Figure 5.16b. Although the drawing is still consistent and satisfies all constraints, the points in (b) have come so close together that we cannot see which points are collinear and which are not. This behavior is very unfortunate since the user will often try to understand the dynamics of a construction by repeating a simple movement over and over again. After all, that is what dynamic geometry systems are for.

During some types of dragging operations, the position of objects will be very sensitive to the cursor movement. For example, many systems allow a free line to be both translated and rotated. In rotation mode, the center of rotation will usually be the position at which the line was picked. Just after the line has been picked, the cursor is close to the center of rotation. A slight shake of the hand will then cause the line to rotate very fast. That will often be enough to make the drawing collapse, if free objects are allowed to drift.

A common cause of drift is that under-constrained objects are moved to the closest allowable position each time the drawing is updated. Because the cursor position is discrete, points on a rotating line will move along short line segments rather than following a circle. The points will follow a spiral ending at the center of rotation, see Figure 5.17. This is the case in Cinderella Café (Section 2.3) for example. The drifting problem is not limited to points on a line. It also exists for movable lines on a point, points on a conic, tangents to a conic etc.

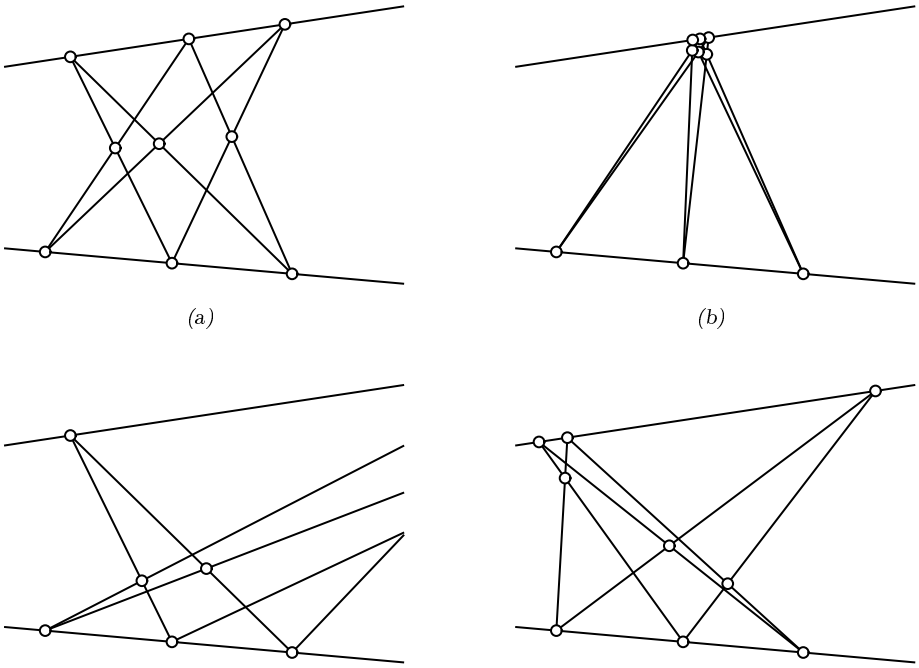


Figure 5.16. Under-constrained points on a line might move unpredictably.

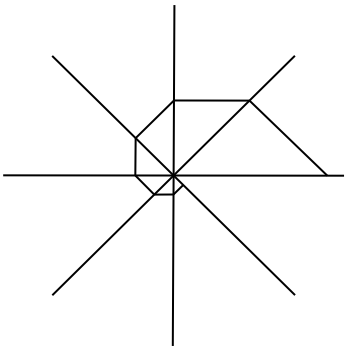
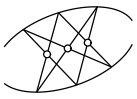


Figure 5.17. If an under-constrained point on a rotating line is always moved to the closest allowable position, it will follow a spiral because of quantization effects.



Floating and runaway objects

Sometimes when a line is dragged, under-constrained points on it quickly slide away along the line and out of sight. This problem was discussed in Section 5.1.2. For example, the slightest movement of the top line in Figure 5.16a may cause some of the points on it to slide away very fast to the right, resulting in the sketch shown in Figure 5.16c.

Even if the points stay visible, they may appear to float around unpredictably on the line. In Figure 5.16d, the top line has just been translated slightly. Still, the distance between the points on it have changed. This behavior will not be intuitive to the user. It makes it more difficult to see how objects are related in the drawing since everything floats around whenever an object is dragged. It also makes it harder to put objects in specific positions since each time an object is moved, another one runs off in another direction.

Note that this is different from the problem of drifting objects. In this case, all objects will return to their original positions if the mouse is returned to the starting position. Nevertheless, if the points on the line are very sensitive to cursor movement, it will be difficult to interact with the drawing.

Typically, this problem occurs if the position of the movable points on the line is determined by (fixed) coordinates in some coordinate system on the line. Depending on how the local coordinate system was chosen and how the line is oriented, the absolute coordinates of the point may change very quickly and unpredictably.

Jumping objects

In Figure 5.18a, a point p has been attached to the intersection of a free line l and a conic C. Suppose that l is dragged. Each time the drawing is updated during a dragging operation, a new position for the intersection point must be computed. In this case, that involves solving a second-degree equation which has two roots (corresponding to the two intersection points). It is important that the system remembers which of the two intersection points that p was attached to. If it fails to keep track of the correspondence between the roots of the system of equations computed in each updating cycle, p will jump unpredictably between the two intersections (Figure 5.18b). That will not only be annoying, but it can also destroy a construction. In Figure 5.18c, another point q has been attached to the second intersection. The intersection of the tangents drawn through p and q represents the pole of l. If p is suddenly given the same position as q, the tangents become identical and the pole disappears (Figure 5.18d).

Figure 5.19a shows a slightly more complicated drawing. It illustrates the fact that the lines connecting the opposite intersection points of two conics and the lines connecting opposite tangent intersections are concurrent. In Figure 5.19b, one of the conics has been dragged slightly. The system has not been able to keep track of the intersection points. Two of them have been interchanged, and connected points are no longer opposite. Consequently, the concurrency in (a) have been destroyed.

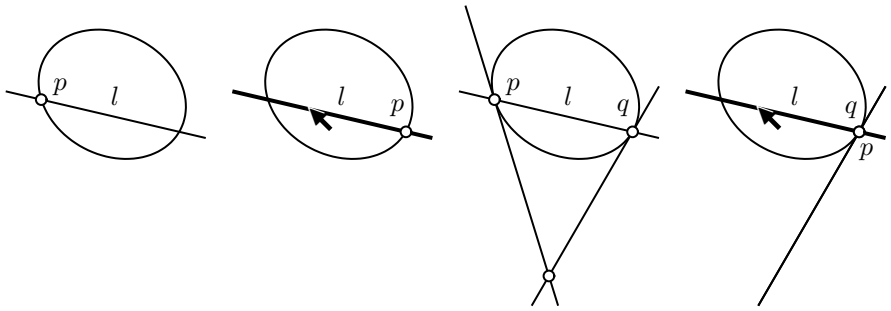


Figure 5.18. If the system does not distinguish the intersection points, the points might jump when the line or the conic is dragged slightly.

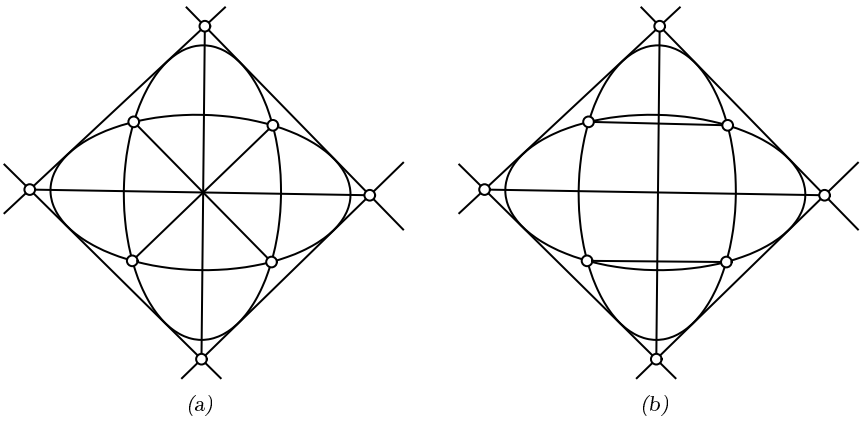
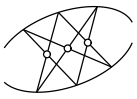


Figure 5.19. Two of the intersection points have been accidently swapped in (b).



Jumping points and lines are usually associated with systems of equations that have a finite set of solutions, but the problem can also be caused by the collapse of a local coordinate system on a line or some similar problem.

5.2.4 Continuity and repeatability requirements

From the observations made above, it is clear that a dynamic geometry system must meet two basic requirements:

- *Repeatability*

  If an object is dragged and returned to its original position, all dependent objects must also return to their original positions. There must be no net movement of any object.

- *Continuity*

  When an object is being dragged, the position of that object and all dependent objects must be continuous functions of the cursor position. No object should be allowed to make sudden jumps.

What are the immediate implications of these requirements? Consider the sequence shown in Figure 5.20. In (a), a line has been attached to a fixed point p. A second point q has been attached to the line. If the line is gripped somewhere to the right of p and the cursor is moved to the left, the line swings around 180 degrees (b-e). Projectively, the position of the line is exactly the same in (a) and (e). Therefore, according to the repeatability requirement, q should have the same position in (a) and (e). Since the motion of q should also be continuous, this means that q must either move in towards p, pass through p and return to the starting position (Figures 5.21a-e), or q must wrap around at infinity (Figures 5.22a-e). Neither alternative is appealing. It means that halfway through the dragging operation, all points on the line must either meet at the center of rotation or, alternatively, go through infinity.

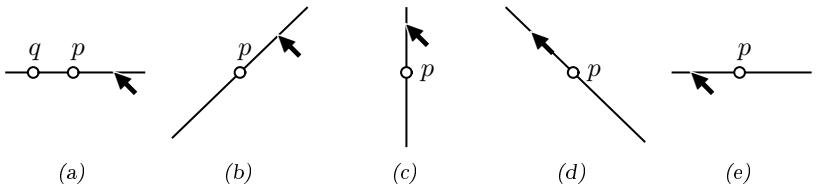


Figure 5.20. When the line is rotated, how should q be moved?

What the user probably thinks should happen is shown in Figures 5.23a-e. The motion of q is continuous, but q does not return to its starting position. To avoid violating the repeatability requirement, we must therefore consider the position of the line in (a) and (e) to be distinct, i.e., we must take the orientation of the line into account. This rules out a purely projective representation of object

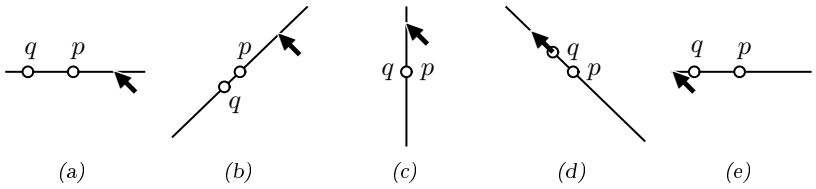


Figure 5.21. If q has to return to its original position in (e), it must pass through p.

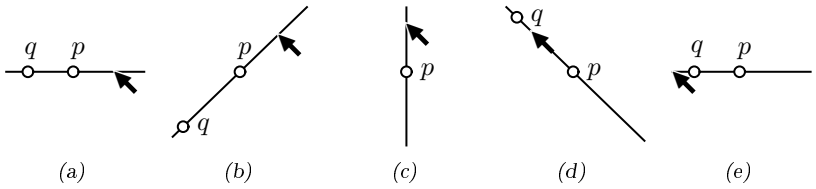


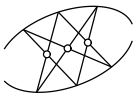
Figure 5.22. Alternatively, q could wrap around at infinity.

positions. What is needed is a representation based on the *oriented* projective geometry discussed in Section 4.13. In oriented geometry, the lines in (a) and (e) are distinct. In fact, they are antipodal. If the dragging operation is repeated, the line will obtain its original orientation and consequently, q should return to its starting position.

It is essential that not only the lines but also the points are given an orientation. Consider the configuration in Figure 5.24. The horizontal line l and the conic is fixed while the point p is movable along l. The position of the tangent is determined by p and the conic. If p moves to the right and passes through infinity[3], it will come back from the left. We see that the position of the tangent is not the same in (a) and (d). That is unavoidable if we want the motion of the tangent to be continuous. If p is sent through infinity again, it is reasonable that we get the original configuration back (e-g). The difference between (a) and (d) must be attributed to the state of p. This fits neatly into the framework of oriented geometry. Each time p wraps around at infinity, it switches between the front and back ranges. If we make sure that the algorithm calculating the tangent takes the orientation of p into account, the effect shown in Figure 5.24 can be achieved.

We conclude that in order to make the geometric constructions stable and object motion continuous, we need more positional information than is available in ordinary (unoriented) projective geometry. We have therefore decided to base the internal representation on an oriented projective framework. However, that does not automatically solve all problems. In the following two sections we will look more closely at how oriented geometry can help us to keep track of intersection points and to make the motion of objects well-behaved.

[3] That cannot be achieved by dragging p. However, p can be attached to the intersection of l and another line m. If m is then made parallel to l, p will go to infinity.



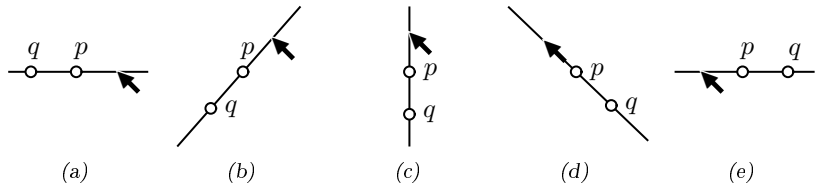


Figure 5.23. If we do not want q to return to its original position in (e), we must keep track of the line orientation.

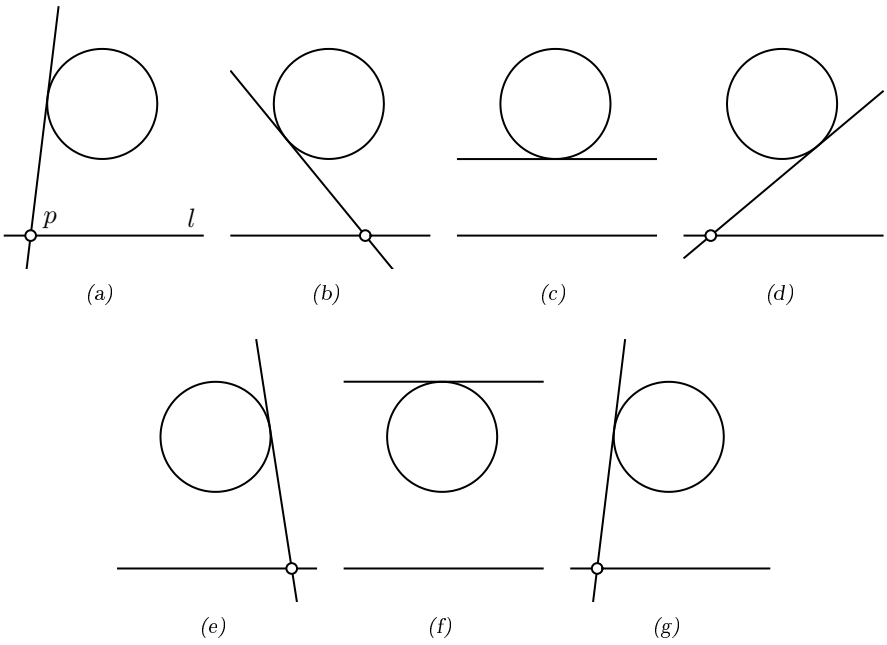


Figure 5.24. When p wraps around at infinity, the tangent switches between the front range and the back range.

5.2.5 Handling multiple roots

The presence of algebraic equations with multiple roots is a common cause of discontinuities in object motion (jumping objects) during dragging operations. Here we will explain why and discuss what can be done about it. The discussion will be based on the conic/line intersection example in Figure 5.18. We will assume that the conic and the line are elements of the oriented projective plane, T_2 (Section 4.13).

At a certain point during the dragging operation, the intersection point in Figure 5.18 jumped into its alternative position. As already noted, this can be avoided if we make sure that the position of the point is a continuous function of the cursor position. But exactly what does "cursor position" and "continuous"

mean? Due to the limited resolution of the pointer device, the cursor movement will be reported to the system as a sequence of discrete coordinates $v_n,\ n = 1, 2, \ldots$. However, the user sees the cursor movement as a continuous curve $\Gamma(t)$. For our purposes, we can use linear interpolation to define that curve in R^2:

$$\Gamma(t) = (n + 1 - t)v_n + (t - n)v_{n+1},\ t \in \mathrm{R},\ n \leq t \leq n + 1$$

During a dragging operation where the cursor is moved along a curve Γ, we will call t the position of the cursor. Let $r(t)$ be the position of the object being dragged or the position of an object depending on the object being dragged. $r(t)$ is a continuous function of the cursor position if

$$\forall t_0 \colon \lim_{t \to t_0 -} \mathrm{dist}(r(t), r(t_0)) = \lim_{t \to t_0 +} \mathrm{dist}(r(t), r(t_0))$$

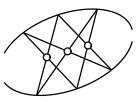
where dist is the distance measure in the metric space we are considering. In R^2, dist may be the usual Euclidean distance. In P_2, the distance between two projective points can be defined as the angle between the R^3 lines they represent in the standard embedding. Such a distance measure is sufficient to define continuity. In T_2, the distance between two oriented points represented by the R^3 vectors p_1 and p_2 can be defined as the Euclidean distance between $p_1/||p_1||$ and $p_2/||p_2||$ on the unit sphere. A point $p(t)$ will then be continuous in T_2 when $p(t)/||p(t)||$ is continuous in R^3.

Let $C(t)$ and $l(t)$ be the position of the conic and the line, respectively, during the dragging operation, and let t represent the position of the cursor. In the following we will assume that l is continuous in T_2 and that C is continuous in T_5, or equivalently, that $l/||l||$ is continuous in R^3 and $C/||C||$ is continuous in $\mathrm{R}^{3\times 3}$. It will then be possible to represent the position of the intersection points by two continuous functions $p(t)$ and $q(t)$. The analytical expressions for $C(t)$ and $l(t)$ are not known in general, but we assume that $C(t)$ and $l(t)$ can be evaluated for any t. Furthermore, we can easily construct an algorithm f which computes the intersections of C and l for any fixed t by solving the following system of equations:

$$\begin{cases} r^T l = 0 \\ r^T C r = 0 \end{cases} \tag{5.11}$$

From this information, we must show how to compute $p(t)$ and $q(t)$ for any value of t, so that the functions become continuous.

Let r_1 and r_2 be the two roots of Equation 5.11 returned by f. In general, f cannot guarantee any specific order of the returned roots. Depending subtly on the implementation of the algorithm, the roots may be interchanged from one call to the next. There can be many reasons for that. To ensure numerical stability, an algorithm may at some point have to choose the largest pivot element or the largest minor for an operation and the outcome of such tests may affect the order of the roots. For example, if f is based on Equation 4.9 on page 37, the choice of q_2 may have to be changed between two calls to make sure it is not too close



to the conic. If f is based on a iterative method, the order in which the roots are found may depend on the seeds or how the search is done. Therefore, for a certain cursor position t_0 we might have

$$\lim_{t\to t_0-} r_1(t) = r_1(t_0), \quad \lim_{t\to t_0+} r_1(t) = r_2(t_0),$$
$$\lim_{t\to t_0-} r_2(t) = r_2(t_0), \quad \lim_{t\to t_0+} r_2(t) = r_1(t_0)$$

Obviously, unless $r_1 = r_2$, r_1 and r_2 are not continuous in $t = t_0$ and can therefore not possibly represent p and q. In principle, we must choose between $p(t) = r_1(t)$ and $p(t) = r_2(t)$ for every t so that p becomes continuous over the whole interval of t. In practice, it is impossible to invoke f for every value of t. f can only be called for a finite set of cursor positions $t = t_1, t_2, \ldots$, typically the ones detected by the windowing system. We are then faced with the problem of determining whether $r_1(t_n)$ corresponds to $r_1(t_{n+1})$ or $r_2(t_{n+1})$. We say that an intersection point r at $t = t_1$ *corresponds* to an intersection point r' at $t = t_2$ if r is continuously transformed into r' when t goes from t_1 to t_2. If we assume that $r_1(t) \neq r_2(t)$ for $t_n < t < t_{n+1}$ then either $r_1(t_n)$ corresponds to $r_1(t_{n+1})$ and $r_2(t_n)$ to $r_2(t_{n+1})$, or $r_1(t_n)$ corresponds to $r_2(t_{n+1})$ and $r_2(t_n)$ to $r_1(t_{n+1})$.

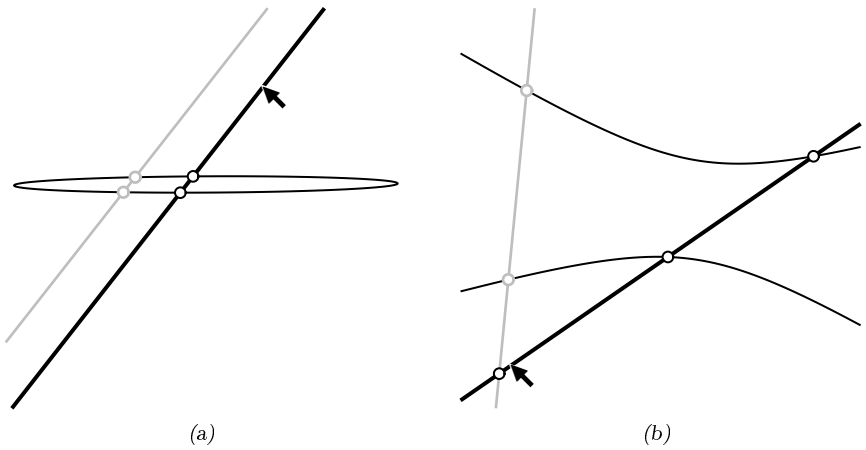


Figure 5.25. Tracking the intersection points is difficult if their movement in each step is large compared to the distance between them.

An simple way of keeping track of corresponding roots is to compare the roots returned by successive calls to f and choose the closest match. We then *assume* that $r_1(t_n)$ corresponds to $r_1(t_{n+1})$ if

$$\mathrm{dist}(r_1(t_n), r_1(t_{n+1})) < \mathrm{dist}(r_1(t_n), r_2(t_{n+1}))$$

and that it corresponds to $r_2(t_{n+1})$ otherwise. This strategy works reasonably well if the step $t_{n+1} - t_n$ is small, but it fails more often than one might expect. For example, if the conic is elongated, the distance between the intersection

points is small compared to the distance that the points move during the dragging operation (Figure 5.25a). Of course, we can split the interval $[t_n, t_{n+1}]$ into several smaller steps and call f and match the roots for each intermediate step. However, in order to choose a suitable step size we must have some idea about how sensitive the roots are to changes in t. If the line l in Figure 5.25b is rotated and the cursor is close to the center of rotation, the intersection points will move a significant distance for each pixel the cursor is moved. To compute the step size in a complicated drawing is not easy. If the step is to small, the response time will be too long.

An alternative is to define a *signature function* s which returns a characteristic value for corresponding roots. If there is only two roots, as in the conic/line intersection example, s could be a real function which always returns a positive value for one root and a negative value for the other. If s is also a *continuous* function of t, then $s(r_1(t_n)) \cdot s(r_1(t_{n+1})) > 0$ if and only if the two roots $r_1(t_n)$ and $r_1(t_{n+1})$ correspond.

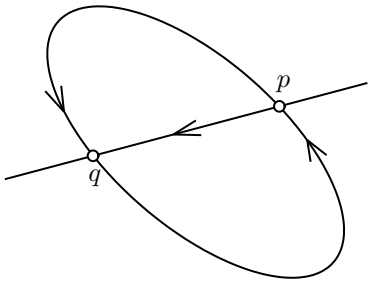
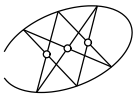


Figure 5.26. An oriented line intersecting an oriented conic.

How do we find such a signature for the conic/line intersections? In T_2, we can use the orientation of the conic and the line, depicted by the arrows in Figure 5.26. If we follow the conic anticlockwise until we reach p, then continue along the line in the direction of the arrow, we will make a left turn. At q, we would instead have made a right turn. This will be true even if the line or conic is transformed continuously. Let us express this fact algebraically.

Suppose that C, l, p, q are continuous and, to begin with, real functions of the cursor position t during the whole dragging operation, and that C is proper for every t. Since p and q are the intersection points, $p^T C p = 0, p^T l = 0, q^T C q = 0, q^T l = 0$. If l becomes tangent to the conic for any value of t, it will not be possible to say anything about the correspondence between intersection points before and after that event. Therefore, we assume that p and q are projectively distinct over the whole interval. Since that implies $p \times q \neq 0$, we can without loss of generality assume that $l = p \times q$ for every t. Let

$$s(p) = \langle l \times Cp, p \rangle$$



With $k = p^T Cq = q^T Cp$ we get

$$\begin{aligned} s(p) &= p^*(l \times Cp) = p^*((p \times q) \times Cp) = p^*((p^T Cp)q - (q^T Cp)p) \\ &= p^*(-kp) = -k|p|^2 \\ s(q) &= q^*(l \times Cq) = q^*((p \times q) \times Cq) = q^*((p^T Cq)q - (q^T Cq)p) \\ &= q^*(kq) = k|q|^2 \end{aligned}$$

$k = 0$ only if p is on the polar of q. If we assume p and q are distinct all the time, $k \neq 0$ for every t. Furthermore, since k is continuous, it cannot change sign. Thus, during the dragging operation, s is strictly positive for one of the intersection points and strictly negative for the other. Recall that the algorithm f for a given t either returns $r_1 = \lambda_1 p$ and $r_2 = \lambda_2 q$ or $r_1 = \lambda_1 q$ and $r_2 = \lambda_2 p$ where λ_1 and λ_2 are any real, non-zero scalars. λ_1 and λ_2 will vary with t and they will *not* be continuous (they will have arbitrary signs).

Now, if $r_1 = \lambda_1 p$ and $r_2 = \lambda_2 q$,

$$\begin{aligned} s(r_1) &= r_1^*(l \times Cr_1) = \lambda_1 \overline{\lambda_1} p^*(l \times Cp) = |\lambda_1|^2 s(p) \\ s(r_2) &= r_2^*(l \times Cr_2) = \lambda_2 \overline{\lambda_2} q^*(l \times Cq) = |\lambda_2|^2 s(q) \end{aligned}$$

On the other hand, if $r_1 = \lambda_1 q, r_2 = \lambda_2 p$,

$$\begin{aligned} s(r_1) &= |\lambda_1|^2 s(q) \\ s(r_2) &= |\lambda_2|^2 s(p) \end{aligned}$$

Since $s(p)$ and $s(q)$ do not change sign, we can detect when r_1 and r_2 are swapped by f. If $c(p) > 0$ and if we define

$$\tilde{p}(t) = \begin{cases} r_1(t), & c(r_1(t)) > 0 \\ r_2(t), & c(r_2(t)) > 0 \end{cases}, \quad \tilde{q}(t) = \begin{cases} r_1(t) & c(r_1(t)) < 0 \\ r_2(t) & c(r_2(t)) < 0 \end{cases}$$

then $\tilde{p}(t) = \lambda_1(t)p(t)$ and $\tilde{q}(t) = \lambda_2(t)q(t)$.

Because we assumed that the roots were distinct, we have $c \neq 0$. If l becomes tangent to the conic, then r_1 and r_2 will represent the same point and $c(r_1) = c(r_2) = 0$. Of course, no unique correspondence can be defined between the roots before and after this singularity, not even in theory. On the other hand, it does not matter which root we choose at that point since they are identical. Therefore, we can simply replace $>$ with $\geq$ in the definition of $\tilde{p}$ above.

The last step in finding the continuous functions p and q is to remove the effect of the scalars λ_1 and λ_2 in $\tilde{p}$ and $\tilde{q}$. The variation in magnitude is no problem; we can always scale $\tilde{p}$ and $\tilde{q}$ to unit magnitude. However, to cancel the effect of a sign change in λ_i, we must use the orientation of the conic. We choose the sign of $\tilde{p}$ and $\tilde{q}$ so that they become consistent with the orientation of the conic. More specifically, if u is the orientation of the conic in T_2 (cf Section 4.13), let

$$\begin{aligned} p &= \eta_1 \langle u, \tilde{p} \rangle \tilde{p} \\ q &= \eta_2 \langle u, \tilde{q} \rangle \tilde{q} \end{aligned} \tag{5.12}$$

where $\eta_1, \eta_2 \in \mathrm{R},\ \eta_1 > 0,\ \eta_2 > 0$ are chosen so that $|p| = |q| = 1$. With this definition we can write

$$\langle u, p\rangle = p^* u = \eta_1 \overline{(\tilde{p}^* u)}(\tilde{p}^* u) = \eta_1 |\tilde{p}^* u|^2 > 0$$

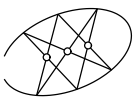
and the same for q.

The use of a signature for identifying the roots has been based on the assumption that the coordinates of the line, the conic and the orientation vector of the conic are all continuous functions of the cursor position t. That must be ensured by the algorithms updating the line and the conic. For example, if the conic is defined by five points and its coefficient matrix C is given by Equation 4.17 on page 44, it is evident that C is continuous only if the coordinates of all five points are continuous. Furthermore, the orientation vector of the conic can be chosen so that one of the five points (the same point all the time) is on the positive side of the double cone in R^3 which corresponds to the conic. Thus, the algorithm we outlined for updating the conic can guarantee continuity if and only if the algorithms updating each of the five points do the same. If every algorithm that updates the position of an object can make similar guarantees then, by induction, the position of every object will be a continuous function of the cursor position.

What if the conic becomes degenerated? If the conic degenerates into a line pair (a rank 2 point conic) the signature s can be used just as before, provided that C is continuous (in T_5) at the singularity. It does not matter that $|C|$ becomes zero as long as the sign of C is not reversed suddenly. For example, if C is defined by Equation 4.17, there will be no such problems. However, as we saw in Section 4.13, a conic cannot maintain a consistent orientation in T_2 if it goes though a degenerated state. The orientation u will be discontinuous at that point. From Equation 5.12 we see that the orientation of p and/or q may be reversed. Thus, it will be possible to distinguish the two intersection points, but we will not be able to give them a consistent orientation. Of course, if the conic degenerates into a double line (a rank 1 point conic), p and q become identical an cannot be distinguished.

So far we have assumed that l intersects C in two real points. However, a real line might also intersect a real conic in two conjugate complex points. Provided that $l \in \mathrm{R}^3$, we may assume that $p = \overline{q}$ and $l = ip \times q$ since $p \times q = \overline{q} \times q$ is purely imaginary. s will then equal $i \cdot const$ where *const* is real and positive for one root and negative for the other. Thus, the intersection points can still be distinguished even though they are conjugate complex. They can also be given a consistent (complex) orientation so that they remain continuous in T_2. Actually, the definition of p and q in (5.12) still works since $\langle u, p\rangle$ and $\langle u, q\rangle$ will be real, positive and continuous even though $\tilde{p}$ and $\tilde{q}$ are complex.

If the line is real but represented by a complex vector (Section 4.14), things get more difficult. In that case, we can only assume that $l = cp \times q$ where c is a complex, non-zero and continuous function of t. Then $s = c \cdot const$ will be complex and its sign will be of no use. If c is varying with t, it is not easy to get rid of it. It effectively destroys the orientation information since it can go from 1 to -1 without passing through 0. However, if the algorithms computing l can somehow



guarantee that c is constant, we may use the signature $s'(p(t)) = s(p(t))/s(p(t_0))$, where t_0 is the starting position of the cursor. Then $s'(p(t_0)) = 1$ and $s'(p(t))$ will be real and positive while $s'(q(t))$ will be real and negative. Of course, the same applies to the conic C.

The algorithm for computing p and q will not work at all if l and C represent general, complex lines and conics since $\mathrm{P}_2(\mathrm{C})$ cannot be oriented (cf Section 4.13). In that case we must resort to continuous tracking.

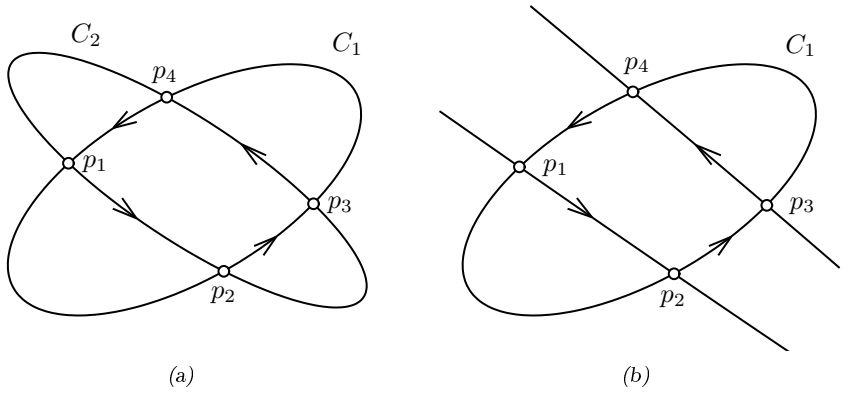


Figure 5.27. Two oriented conics.

The approach just described works well for conic/line intersections. Naturally, it can also be used for the dual problem of computing the tangents of a conic through a given point. However, if there are more than two roots, this approach does not give a complete solution. For example, two conics C_1 and C_2 have four common points p_1, p_2, p_3, p_4 (not necessarily distinct or real), see Figure 5.27a. An algorithm solving the system

$$\begin{cases} p^T C_1 p = 0 \\ p^T C_2 p = 0 \end{cases}$$

will not be able to guarantee any specific order of the roots returned. As before, we must find the correspondence between roots at cursor position t_n and t_{n+1}. However, in this case the orientation of the conics will not provide enough information to do that. If we move along C_1 anticlockwise to p_1 in Figure 5.27a, then anticlockwise along C_2, we will make a left turn. At p_2, p_3 and p_4 we would instead have made a right turn, a left turn, and a right turn, respectively. Thus, we can distinguish p_1 from p_2 and p_4, but not from p_3. Similarly, p_2 can be distinguished from p_1 and p_3 but not from p_4. That also becomes evident if we let C_2 degenerated into a line pair as in Figure 5.27b. The orientation of the lines is determined by the orientation of the conic being degenerated. Using a signature for the conic/line intersections we can distinguish p_1 from p_2 and p_3 from p_4. On the other hand, if we instead let C_1 degenerate in the same way, we can

distinguish p_1 from p_4 and p_2 from p_3. Suppose that the conics in Figure 5.27a are centered at the origin. The projectivity

$$M = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

which represents a 180 degrees rotation will map both conics onto themselves while preserving their orientation. Still, M will swap both p_1, p_3 and p_2, p_4. This shows that we cannot even in principle use the orientation of the conics to distinguish p_1 from p_3 and p_2 from p_4.

For the intersection of two conics, the orientation of the conics can be used for distinguishing neighboring intersections, while opposite intersections must be continuously tracked using a sufficiently small step size $|t_{n+1} - t_n|$. The orientation information is still very valuable since neighboring intersections often comes very close together, making them hard to track reliably.

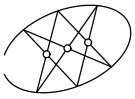
We should mention that there are ways of circumventing the problem of multiple roots. In Cinderella Café (Section 2.3) for example, it is not possible to pick out a *single* tangent to a conic. Instead, both tangent lines are constructed at once, and they behave like a degenerated conic (a line pair). It is not possible to distinguish the two tangents as they are topologically connected; a point placed on one tangent might migrate to the other line if the shape of the conic is modified. We feel that this is counter-intuitive and prevents the user from creating many interesting drawings.

5.2.6 Controlling the motion of under-constrained objects

In Section 5.2.4 we argued that continuous object motion during dragging is a highly desirable feature of a dynamic geometry system. In the previous section we looked at a common cause of discontinuities, namely equations with multiple solutions, and showed how that could be handled. We will now address the remaining problems discussed in Section 5.2.3, namely floating objects (incoherent motion) and drift (lack of repeatability). We will outline an algorithm for updating the drawing during dragging operations which provides both continuous object motion and repeatability in most situations, and which can be implemented efficiently.

General requirements

During a dragging operation, it is straight-forward to update an object whose position is completely determined by user-defined constraints. We just have to make sure that its coordinates are not recomputed until all objects on which it depends have been updated. Such an updating order can always be found since the dependency graph is acyclic (Section 5.1.4). Furthermore, to satisfy the continuity requirement, the position of the object must be a continuous function of the position of its parents. Completely unconstrained objects pose no problem



either. They have to be repositioned only if they are dragged and in that case, they are free to follow the movements of the cursor.

The problem is how to handle under-constrained objects. An under-constrained object has to be repositioned if either the object is dragged or any object on which it depends is moved. Additional constraints are needed to completely specify the path that should be taken by the object during a dragging operation. As noted in Section 5.1.2, the choice of such constraints is to some extent arbitrary. Apart from the continuity and repeatability requirements discussed in Section 5.2.4, the following three constraints seem reasonable:

1. If the under-constrained object *itself* is dragged, it should minimize the distance to the cursor.

2. Otherwise, if possible, the object should move in the same way as the rest of the objects. The motion on the screen will be more coherent if as many objects as possible move in the same direction at the same speed or if they rotate around the same point.

3. The Euclidean distance between moving objects should be preserved, if that is possible with regard to the user-defined constraints. The moving parts of the drawing will then appear to be rigid and they will clearly stand out from the background of stationary objects. It is easier for the user to follow the motion of a large rigid structure than to keep track of many small objects moving in different directions. If the constraints on the objects do not allow for rigid motion, angle preserving motion is the second best alternative. The size of the moving object configuration will then be affected, but not the shape.

Before we elaborate on this, we have to define what it means for two object to "move in the same way". As in the previous section, let $\Gamma(t)$, $t \in \mathrm{R}$, be the path of the cursor during a dragging operation, parameterized by arc length. The dragging operation is initiated at $t = 0$, and t will be referred to as the position of the cursor. If p is a point being dragged or a point depending on an object being dragged, its position will be a function of the cursor position t. Because we are interested in coherent motion, we will be concerned primarily with the way objects move relative to each other and not so much with the path $p(t)$ of an individual object. We will therefore represent the motion of a point p by a projectivity $M(t)$ such that

$$\forall t\colon p(t) = M(t)p(0)$$

Thus, for any cursor position t, $M(t)$ specifies how p has moved relative to its starting position. We can then say that two points p and q move in the same way if

$$\forall t\colon p(t) = M(t)p(0),\ q(t) = M(t)q(0)$$

Furthermore, if $M(t)$ is an isometry for each t, then the distance between p and q will be the same for all t, i.e.

$$\forall t\colon \operatorname{dist}(p(t), q(t)) = \operatorname{dist}(M(t)p(0), M(t)q(0)) = \operatorname{dist}(p(0), q(0))$$

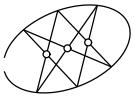
What is expressed by requirements (2) and (3) above is that we should select the paths of the under-constrained objects so that their positions can be described by the same motion M, and that $M(t)$ should preferably be a Euclidean isometry (rigid motion) or a similarity transformation for each t. According to the continuity requirement, the path $p(t)$ of an under-constrained object has to be continuous in T_2. This implies that $M\colon \mathrm{R} \to \mathrm{R}^{3\times 3}$ must be a continuous function of t. Moreover, in order to fulfill the repeatability requirement, if the cursor is returned to its starting position so that if $\Gamma(t_1) = \Gamma(0)$ for some value t_1, then $M(t_1)$ must be the identity transformation .

If all under-constrained objects cannot move in the same way, then they should be divided into groups so that the members of each group can be given the same motion. How can this be accomplished? If there are only incidence constraints on an under-constrained object x, and if M is the motion of all parents of x, then we can let M be the motion of x also. This follows from the fact that any projectivity preserves incidences. In particular, if x is required to be incident on exactly one other object, x can always be given the same motion as that object. If there are metric constraints on x, M is an allowable motion for x only if M is an isometry.

Thus, we can adopt the following strategy: A motion $M_1(t)$ is specified for the object being dragged. That motion is inherited by the descendants in the dependency graph as far down as possible. If the motion of the parents of an object x is not compatible with the constraints on x, or if not all parents have the same motion, then a new motion $M_2(t)$ has to be specified for x. That motion will in turn be inherited by the descendants of x, and so on. When a new motion is defined for an object, we prefer the simplest form that is compatible with existing constraints:

1. translation or rotation
2. rigid motion (a combination of translation and rotation)
3. similarity transformation (angle preserving)
4. general projectivity

As we shall see in Section 5.3.1, the user may have several open windows, which show different projections of the same drawing. When we talk about translations, rotations and similarity transformations, we are referring to the Euclidean metric implicitly defined by the coordinate system of the window in which the user interaction occurs. In general, it will not be possible to have a set of objects move rigidly in all windows at the same time – we have to make a choice.



Motion of specific objects

We now make a precise specification of the motion of every type of object, taking into account the general requirements discussed above. *For objects being dragged* (objects which of course must be under-constrained) we define the following rules:

- A free point and a point on a line or on a conic is always translated, i.e., $M(t)$ is a translation for every t. The translation of a free point is the same as that of the cursor: $\Gamma(t) = M(t)\Gamma(0)$.
- The motion of a free line is either the translation of the cursor, M: $\Gamma(t) = M(t)\Gamma(0)$, or a rotation around $\Gamma(0)$ depending on the operating mode (the tool currently active, state of modifier keys etc).
- A line on a point is rotated around that point.
- The motion of a line on a conic is described by a rigid motion. For each cursor position t, $M(t)$ consists of a translation from $\Gamma(0)$ to $\Gamma(t)$ followed by a rotation which makes the line tangent to the conic.

For objects that are not dragged but which have to be moved because another object is dragged, we define the following rules:

- An object with only one parent in the dependency graph (such as a point on a line, a line on a point, a point on a conic, and a line on a conic) is given the same motion as its parent.
- The motion of a point p representing the intersection of two lines, a line and a conic or two conics, is defined as the translation $M(t)$ satisfying $p(t) = M(t)p(0)$, provided that neither $p(0)$ nor $p(t)$ is at infinity.
- The motion of a line on two points p and q is defined as a similarity transformation $M(t)$ such that $p(t) = M(t)p(0)$ and $q(t) = M(t)q(0)$. Thus, the line is translated and rotated but is also "stretched" if p and q move further apart.
- The motion of a line on a point p and a conic C is rigid. It is a combination of the translation of p and the rotation around p necessary to keep the line tangent to C.
- A common tangent of two conics is moved as a line on two points (see above), where the two points are the tangent points to the conic.
- A line l with a fixed angle to a reference line m is given the same motion as m. If also the given angle value is changed, this motion is followed by a rotation around the intersection point of l and m.
- The motion of a line on a point with a fixed angle to a reference line is the translation of the point followed by the rotation needed to keep the angle constraint satisfied.

- A point at a fixed distance from another point (possibly attached to a line) is always translated.

Any object that does not need to be moved will not be moved. In particular, completely unconstrained objects will not move unless they are dragged. For stationary objects, $M(t) = E$ for all t.

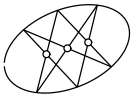
Let us look at an example to see how this algorithm works. The construction in Figure 5.28a illustrates the theorem of Desargues. The two triangles $v_1v_2v_3$ and $v_1'v_2'v_3'$ are perspective from p, and the intersections of corresponding sides are therefore concurrent (cf Section 4.6). Figure 5.28b shows the corresponding dependency graph. The top node represents p. Three lines l_1, l_2, l_3 have been attached to p, and two points v_i, v_i' have been placed on each line. The sides of the two triangles are represented by `LineOnTwoPoints` nodes. Two of the intersections of corresponding sides, q_1 and q_2, are represented by `PointOnTwoLines` nodes. Finally, the line between q_1 and q_2 has been defined as a `LineOnTwoPoints` at the bottom of the dependency graph. Now, if p is dragged, its motion will be described as a translation which will be propagated downwards. The lines on p and the vertices of the triangles have only one parent, so they inherit this motion. Since the sides of the triangles are `LineOnTwoPoints` their motion is described by similarity transformations. However, since all vertices are translated in the same way, the similarity transformations will in this case represent the same translation. The same is true for the rest of the objects. Thus, the whole construction is translated like p. If l_2 is dragged instead, it will rotate around p, see Figure 5.28c. That rotation will be inherited by v_2 and v_2', and the distance between v_2, v_2' and p will be preserved. The four affected sides v_1v_2, v_2v_3, $v_1'v_2'$ and $v_2'v_3'$ define similarity transformations which are propagated to the two intersection points q_1 and q_2. The relative distances between v_1, v_2, q_1 etc are therefore preserved.

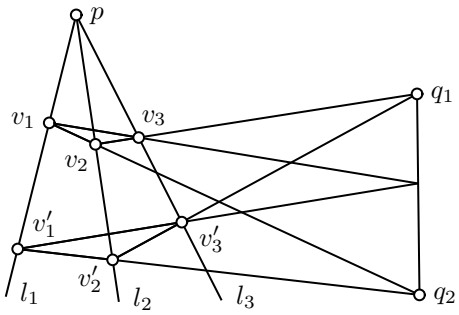
We have not yet discussed how the motion of a conic should be computed. The motion information will be used for updating points and lines attached to the conic. If the coefficient matrix of the conic is $C(t)$, then for each cursor position t we need to find a projectivity $M(t)$ which maps $C(0)$ to $C(t)$. That is

$$\forall t\colon C(t) = M^{-T}(t)C(0)M^{-1}(t)$$

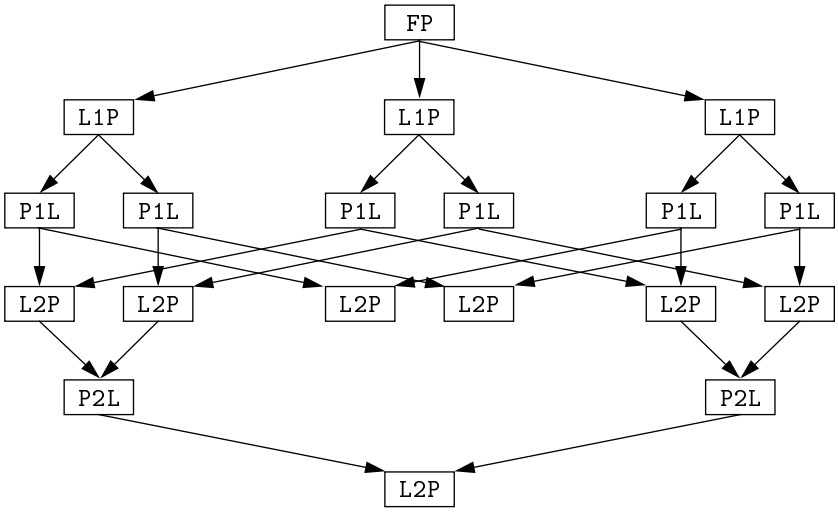
Furthermore, M should be a continuous function of t and $M(0) = E$. If $M_1(t)$ projects $C(t)$ onto the unit circle, then we can choose $M(t) = M_1^{-1}(t)M_1(0)$. In Section 4.8.7, we showed how M_1 can be computed through diagonalization and scaling. However, M_1 will not be uniquely determined by the diagonalization process. The problem is basically that the unit circle can be mapped onto itself by an arbitrary rotation. Therefore, we define M_1 as follows.

Let r be the orientation of the conic. Let Π be the R^3 plane which contains the z-axis and r, and let n be a normal vector of Π. Let R_1 be the rotation in R^3 whose axis is n and which maps r onto the z-axis. Since both r and the axis are directed in oriented geometry, R_1 is uniquely determined. R_1 turns the conic into an ellipse, centered at the origin. Next, determine a rotation R_2 in the projective plane which aligns the major and minor axis of the ellipse with the x-

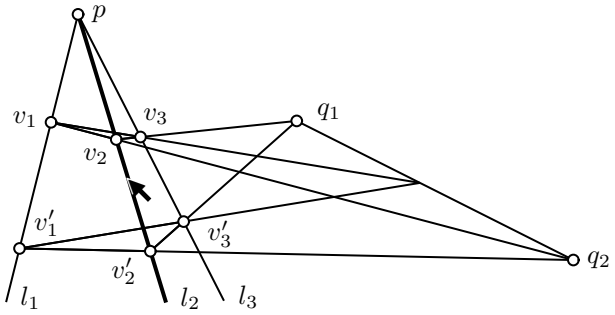




(a)



(b)



(c)

Figure 5.28. Updating a sketch showing the theorem of Desargues. Legend: FP=free point, P1L=point on one line, P2L=point on two lines, L1P=line on one point, L2P=line on two points.

and y-axis. (This rotation is *not* uniquely determined.) Find a matrix S which scales the coefficients of the rotated ellipse to unit magnitude. Finally, rotate the ellipse back again with R_2^{-1}, since that rotation was arbitrary. The resulting transformation is $M_1 = R_2^{-1} S R_2 R_1$, and it maps C onto the unit circle.

A problem with this approach is that the orientation r of the conic is discontinuous when C goes through a degenerated state. We need to do some continuous tracking to detect that. Another flaw is that the points on the conic can move a significant distance even though the cursor is moved very little.

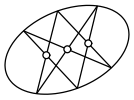
An alternative is to move every point on C to the position closest to its current position. That is, we let $p(t)$ be the point on $C(t)$ that is closest to $p(t - \Delta t)$. However, this will cause the points on C to drift: if the shape of an ellipse is varied, the points on it will typically move closer and closer together. To avoid drift, we could let $p(t)$ be the point on $C(t)$ that is closest to $p(0)$. Then all points on C will return to their original positions if the cursor is moved back to $\Gamma(0)$. In this case, the problem is that $p(t)$ is not guaranteed to be continuous. At a certain point a completely different point on C can become closer to $p(0)$, and then the point p will appear to jump.

Verifying the continuity and repeatability requirements

We noted above that $\Gamma(t) = \Gamma(0)$ should imply $M(t) = E$. It is easy to see that the rules for computing the motion above guarantee that: M expresses the relation between the current position of an object and the position it had when the dragging operating was initiated. In principle, the only input to the updating algorithm is the original positions of all objects and the current position of the cursor.

However, there is no such guarantee for a series of successive dragging operations. For example, if a point is dragged from a position p, dropped at p', picked up again, and moved back to p, the positions of one or several objects may have changed. That usually happens in complicated drawings where objects wrap around at infinity and conics go through degenerate states.

To determine the motion of objects according to the rules above we will need to compute, among other things, the rigid motion which maps a line $l(0)$ onto $l(t)$ so that a point $p(0)$ on $l(0)$ corresponds to a point $p(t)$ on $l(t)$. Conceptually, we first translate $l(0)$ so that $p(0) \mapsto p(t)$, and then rotate the translated line so that it becomes aligned with $l(t)$. In unoriented geometry, this rotation is not uniquely determined; if two unoriented lines can be aligned by a rotating α, they can also be aligned by a rotation $\pi - \alpha$ in the opposite direction. These two rotations represent completely different projective transformations, even in unoriented geometry. There is an obvious risk that we will choose a different rotation each time the drawing is updated, which will make $M(t)$ discontinuous. Again, the result will be jumping objects. Luckily, there is no such ambiguity problem in oriented geometry, since every line has a direction. The rigid motion is completely determined by $p(0)$, $l(0)$, $p(t)$ and $l(t)$. Likewise, if we need to determine the similarity transformation which maps $p(0)$ to $p(t)$ and $q(0)$ to $q(t)$, we will compute the



unique projectivity which maps $p(0), q(0), I, J$ to $p(t), q(t), I, J$. If $p(t)$ and $q(t)$ are oriented and continuous in T_2, $M(t)$ will also be continuous in T_2.

Objects with complex coordinates

The updating algorithm we have outlined has to work also for objects with complex coordinates. The rules for computing object motion defined above can in principle be used for complex objects, although the result will be a *complex* motion matrix $M(t)$. However, there are certain problems associated with complex coordinates that have to be addressed.

First, it is not obvious what should happen when an object with complex coordinates is dragged. We suggested above that it should minimize the distance to the cursor, but the Euclidean distance between a complex point and the (real) cursor position will be complex. We could instead require that the distance between the real screen image of the point (cf Section 5.3.4) and the cursor should be minimal, but that will not suffice to completely define the position of the point. The basic problem is of course that the coordinates of a complex point consists of four real numbers, while the cursor provides only two. Because of this, the user is not allowed to drag objects with complex coordinates in pdb. Nevertheless, these objects have to be moved during dragging operations if they depend on the (real) object being dragged. Therefore, we must still be able to compute the motion $M(t)$ for complex objects.

We saw above that the motion $M(t)$ of every object could be made continuous in the real case, thereby satisfying the continuity requirement. A necessary condition was that $M(t)$ could be determined *unambiguously* from the starting positions of the objects and the current cursor position. In the case of rigid motion, we relied on the fact that every line has a well-defined orientation in T_2. That is not true in the complex case since the complex projective plane has no front and back range (see Section 4.13). Therefore, let us take a closer look at how a rigid motion and a similarity transformation can be computed, and what problems occur in the complex case.

As explained in Section 4.12.3, a similarity transformation leaves the circular points I and J invariant. It follows that a similarity transformation is completely defined by two pairs of corresponding points p_1, p_2 and q_1, q_2, since there is a unique projectivity which maps four points p_1, q_1, I, J to four other points p_2, q_2, I, J. This projectivity can be computed as described in Section 4.3. No problem arises if p_1, p_2, q_1 or q_2 happen to be complex.

For a rigid motion, the situation is a little different. Let us first see how it can be computed in the real case. We want to find the rigid motion M which maps a line l_1 to l_2 and a point p_1 on l_1 to a point p_2 on l_2, see Figure 5.29. Since we can anticipate problems with trigonometric functions in the complex case, we avoid using them. That will also speed up the calculations. We assume that p_1 and p_2 are not at infinity and that neither l_1 nor l_2 is the line at infinity. In principle, we first translate p_1 and l_1 so that p_1 is moved to the origin, then rotate the translated line so that it becomes aligned with l_2. Finally, we translate the line

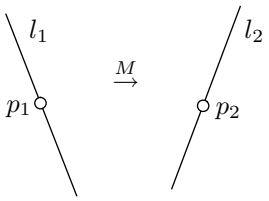


Figure 5.29. A rigid motion.

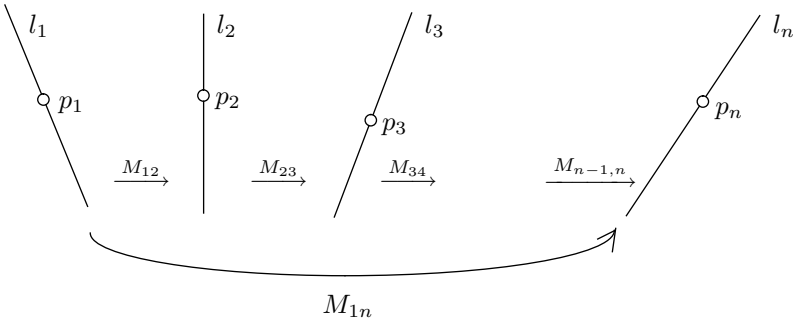


Figure 5.30. Successive rigid motions.

onto l_2 so that the origin is mapped to p_2. Thus, the rigid motion can be written $M = T_j^{-1} R_{ij} T_i$, where T_i is the translation which takes p_i to the origin, and R_{ij} is the rotation which aligns the translated lines $T_i^{-T} l_i$ and $T_j^{-T} l_j$. If

$$p_i = \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix}, \; l_i = \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix}$$

then

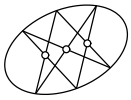
$$T_i = \begin{pmatrix} 1 & 0 & -x_i \\ 0 & 1 & -y_i \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$T_i^{-T} l_i = \begin{pmatrix} a_i \\ b_i \\ 0 \end{pmatrix}$$

R_{ij} represents a Euclidean rotation which can be written

$$R_{ij} = \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; c^2 + s^2 = 1$$



(cf Section 4.12.3). We require that

$$R_{ij}^{-T} T_i^{-T} l_i = k T_j^{-T} l_j, \; k \in \mathrm{C}$$

which can be written

$$\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \\ 0 \end{pmatrix} = k \begin{pmatrix} a_j \\ b_j \\ 0 \end{pmatrix}$$

where k is chosen so that $c^2 + s^2 = 1$. Since the equation corresponding to the last row does not constrain k, we can write

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_i \\ b_i \end{pmatrix} = k \begin{pmatrix} a_j \\ b_j \end{pmatrix} \tag{5.13}$$

Since

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c \\ s \end{pmatrix} = \begin{pmatrix} -s \\ c \end{pmatrix}$$

the left-hand side of Equation 5.13 can be written

$$a_i \begin{pmatrix} c \\ s \end{pmatrix} + b_i \begin{pmatrix} -s \\ c \end{pmatrix} = a_i \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c \\ s \end{pmatrix} + b_i \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c \\ s \end{pmatrix} = \begin{pmatrix} a_i & -b_i \\ b_i & a_i \end{pmatrix} \begin{pmatrix} c \\ s \end{pmatrix}$$

and we get

$$\begin{pmatrix} a_i & -b_i \\ b_i & a_i \end{pmatrix} \begin{pmatrix} c \\ s \end{pmatrix} = k \begin{pmatrix} a_j \\ b_j \end{pmatrix}, \; k \in \mathrm{C}$$

If we define

$$A_i = \begin{pmatrix} a_i & -b_i \\ b_i & a_i \end{pmatrix}, \; \mu_i = a_i^2 + b_i^2$$

then

$$\begin{pmatrix} c \\ s \end{pmatrix} = k A_i^{-1} \begin{pmatrix} a_j \\ b_j \end{pmatrix} = \frac{k}{\mu_i} \begin{pmatrix} a_i & b_i \\ -b_i & a_i \end{pmatrix} \begin{pmatrix} a_j \\ b_j \end{pmatrix} = \frac{k}{\mu_i} \begin{pmatrix} a_i a_j + b_i b_j \\ a_i b_j - b_i a_j \end{pmatrix}$$

This gives us

$$c^2 + s^2 = \frac{k^2 \mu_j}{\mu_i}$$

$c^2 + s^2 = 1$ implies

$$k = \pm \frac{\sqrt{\mu_i}}{\sqrt{\mu_j}}$$

The choice of sign of k will determine the sign of the 2×2 upper-left sub-matrix of R_{ij}. These two possibilities correspond to a rotation α in one direction and a rotation $\pi - \alpha$ in the opposite direction, provided that the angle is real. If everything is real, we can simply choose the positive value of k, which means that the left-hand and right-hand sides in Equation 5.13 will represent the same line in T_2. Then

$$\begin{pmatrix} c \\ s \end{pmatrix} = \frac{1}{\sqrt{\mu_i \mu_j}} \begin{pmatrix} a_i a_j + b_i b_j \\ a_i b_j - b_i a_j \end{pmatrix}$$

and

$$R_{ij} = \begin{pmatrix} \lambda_{ij} Q_{ij} & 0 \\ 0 & 1 \end{pmatrix}$$

where

$$\lambda_{ij} = \frac{1}{\sqrt{\mu_i \mu_j}}, \; Q_{ij} = \begin{pmatrix} a_i a_j + b_i b_j & b_i a_j - a_i b_j \\ a_i b_j - b_i a_j & a_i a_j + b_i b_j \end{pmatrix}$$

However, if l_i and l_j are complex, μ_i and μ_j will be complex and the sign of $\sqrt{\mu_i \mu_j}$ will be undetermined. If we just pick an arbitrary sign, M will not necessarily be a continuous function of p_1, l_1, p_2, l_2. If t is the cursor position during dragging, $M(t)$ might rotate l_1 in one (complex) direction, while $M(t + \Delta t)$ rotates the line in the opposite direction. We can avoid this potential discontinuity by choosing, for example,

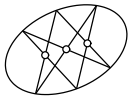
$$\lambda_{ij} = \frac{|\sqrt{\mu_i \mu_j}|}{\mu_i \mu_j}$$

If μ_i and μ_j are real, R_{ij} will not be affected by this change. The advantage is that this factor is well-defined even in the complex case. However, if μ_i and μ_j are complex, $c^2 + s^2 \neq 1$ and R_{ij} will not be a rotation exactly. As far as the user interaction is concerned, that will be acceptable. If the lines are complex, their (real) images of the screen will probably not appear to rotate even if $c^2 + s^2 = 1$ (cf Section 5.3.4).

However, there is another problem that has to be considered when selecting the scalars λ_{ij}. Suppose that the point p and the line l are moved to positions $p_1, p_2, \ldots, p_n$ and $l_1, l_2, \ldots, l_n$, respectively, by a series of successive dragging operations. The user might be dragging p, l or any other object on which p and l depend. As a concrete example, assume that p is a free point being dragged, and that l is a line on p which is also tangent to a given conic. At each position p_i, the point is dropped and picked up again. Let the rigid motion determined by p_i, l_i, p_j, l_j for each separate dragging operation be M_{ij}, that is,

$$p_j = M_{ij} p_i, \; l_j = M_{ij}^{-T} l_i$$

(see Figure 5.30). Compare that with a scenario where p is dragged directly from position p_1 to p_n in one operation. In the first case, the accumulated motion is



$M_{n-1,n} \dots M_{23} M_{12}$. We would like this to be equal to the motion in the second case, M_{1n}. Why? In the updating algorithm outlined above, M_{ij} will describe the motion of the line l. It will be applied to all under-constrained points attached to l and possibly their descendants. If $M_{n-1,n} \dots M_{23} M_{12} \neq M_{1n}$, the final position of these under-constrained objects will depend on whether p is displaced by a single dragging operation. What is even worse is that $M_{n-1,n} \dots M_{23} M_{12}$ might be complex if some of the intermediate line positions $l_2, \dots l_{n-1}$ are complex, even though the starting position and final position, l_1 and l_n, are real. That would mean that an under-constrained point on l that was real before the series of dragging operations could end up in a complex position on l afterwards, even though both l_1 and l_n are real. We do not want this to happen since pdb should try to keep the coordinates of an under-constrained object real, if that is possible with regard to the given constraints on the object (cf Section 5.2.2).

First, we note that the fact that a projectivity maps a real line onto a real line does not mean that the projectivity has to be real. For example, the complex projectivity

$$\begin{pmatrix} i & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

maps the real line $(0,0,1)^T$ onto itself, but maps one of its real points $(1,1,0)^T$ to $(i,1,0)^T$. To understand how the choice of λ_{ij} affects $M_{n-1,n} \dots M_{23} M_{12}$, we must take a closer look at the expression of M_{ij}.

$$\begin{aligned} M_{n-1,n} \dots M_{23} M_{12} &= T_n^{-1} R_{n-1,n} T_{n-1} \dots T_3^{-1} R_{23} T_2 T_2^{-1} R_{12} T_1 \\ &= T_n^{-1} R_{n-1,n} \dots R_{23} R_{12} T_1 \end{aligned}$$

Since all translations except the first one and the last one cancel, we can concentrate on the concatenated rotation matrices, $S_{1n} = R_{n-1,n} \dots R_{23} R_{12}$. It is easily verified that

$$Q_{mj} Q_{im} = \mu_m Q_{ij}, \ i < m < j$$

Thus

$$\begin{aligned} S_{1n} &= \begin{pmatrix} \lambda_{n-1,n} Q_{n-1,n} & 0 \\ 0 & 1 \end{pmatrix} \dots \begin{pmatrix} \lambda_{23} Q_{23} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \lambda_{12} Q_{12} & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \lambda_{n-1,n} \dots \lambda_{12} Q_{n-1,n} \dots Q_{12} & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \eta Q_{1n} & 0 \\ 0 & 1 \end{pmatrix} \end{aligned}$$

where

$$\eta = \lambda_{12} \lambda_{23} \dots \lambda_{n-1,n} \cdot \mu_2 \mu_3 \dots \mu_{n-1}$$

We see immediately that with the original choice of λ_{ij}, namely $\lambda_{ij} = 1/\sqrt{\mu_i \mu_j}$,

$$\eta = \frac{\mu_2 \mu_3 \dots \mu_{n-1}}{\sqrt{\mu_1}\sqrt{\mu_2}\sqrt{\mu_2}\sqrt{\mu_3} \dots \sqrt{\mu_{n-1}}\sqrt{\mu_n}} = \frac{1}{\sqrt{\mu_1 \mu_n}} = \lambda_{1n}$$

Hence we have

$$S_{1n} = R_{1n}$$

We can also see that if

$$\lambda_{ij} = \frac{|\sqrt{\mu_i \mu_j}|}{\mu_i \mu_j}$$

the factors in η will not cancel and $S_{1n} \neq R_{1n}$. A suitable choice is therefore

$$\lambda_{ij} = \frac{|\sqrt{\mu_i}|}{\mu_i |\sqrt{\mu_j}|} \tag{5.14}$$

That will satisfy all of our requirements:

- $S_{1n} = R_{1n}$
- R_{ij} is continuous, even if p_i, l_i, p_j, l_j are complex.
- R_{ij} will be a true Euclidean rotation if p_i, l_i, p_j, l_j are real.

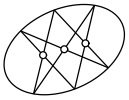
Therefore, all objects which describe their motion as rigid will use Equation 5.14 for computing λ_{ij}.

5.2.7 Representing metric information

In Sections 4.11 and 4.12, we discussed how angles and distances can be defined and showed how different definitions lead to different types of geometries. We noted that concepts such as midpoints, angle bisectors, focal points, etc. depend on the metric we choose.

Since distances and angles can be defined in more than one way, and since they are not preserved by general projective transformations, we have been careful not to use metric concepts in classes that represent purely projective elements such as points, lines, conics, and incidence constraints. A class which makes implicit assumptions about a particular metric (typically a Euclidean metric) will not be useful when we explore geometries based on other metrics. For example, we might be tempted to introduce a *line segment* primitive. The segment between two points p and q could be defined as the set of points r such that p and q are separated (Section 4.9) by r and the real infinity point on the line. However, "infinity point" is a metric concept, i.e., we need a metric to define it. The line may have two real infinity points or none at all, depending on the metric. When we talk about *the* infinity point, we have implicitly assumed a geometry with a parabolic distance measure, such as Euclidean geometry.

Therefore, metric concepts have been added on top of pdb's projective elements, as a set of new node types which represent angles, distances, metrics and metric constraints. Some of the nodes representing metric constraints, such as `LineOnPointWithAngle`, were mentioned briefly in the previous section. Angle



and distance measurements are represented by `Angle` and `Distance` nodes, respectively. The metric itself is represented by a `Metric` node. In this section, we will explain how these nodes cooperate and how angle and distance values are represented and used in the dependency graph.

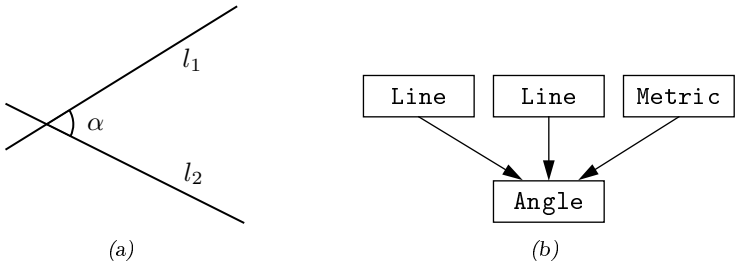


Figure 5.31. Measuring the angle between two lines.

In Figure 5.31a, we have measured the angle α between two lines l_1 and l_2. The corresponding dependency graph is shown Figure 5.31b. The `Angle` node represents α and the two `Line` nodes represent l_1 and l_2, respectively. The third parent, the `Metric` node, specifies how the angle is calculated. In principle, a `Metric` node encapsulates the formulas for computing angles and distances. Different `Metric` subclasses represent different metrics and thus contain different formulas. For example, there is a `EuclideanMetric` subclass which computes Euclidean distances and angles. If the user moves one of the parent lines or chooses another metric in Figure 5.31a, the `Angle` node will immediately update the angle α.

An angle value can also be used in metric constraints. In Figure 5.32a, there is a constraint on m_2 which makes its angle to m_1 equal to α. If any of the lines l_1, l_2 or m_1 is dragged, the position of m_2 will be updated to satisfy the constraint, as shown in Figure 5.32b. The corresponding dependency graph is shown in Figure 5.32c, where the `LineWithAngle` node represents m_2. The `Line` parent of `LineWithAngle` represents the base line from which β is measured, in this case m_1. The α value is fetched from the `Angle` node, and from that information `LineWithAngle` calculates the orientation of m_2. Note that the `Angle` and `LineWithAngle` nodes must use the same `Metric` to make the interpretation of the angle value consistent. If we want to verify that the angle between m_1 and m_2 really equals α, we can add a new `Angle` node as shown in Figure 5.32d. The corresponding drawing is shown in Figure 5.32e.

In Figure 5.32c, we saw that angle values must be passed between nodes in the dependency graph, which brings up the question of *angle representation*. In high school geometry, an angle is simply a real scalar. However, there are several problems and ambiguities associated with such a representation:

- The angle is usually only defined modulo π.
- How do we distinguish an angle from its complement? Is the angle between lines l_1 and l_2 in Figure 5.33, α or β?

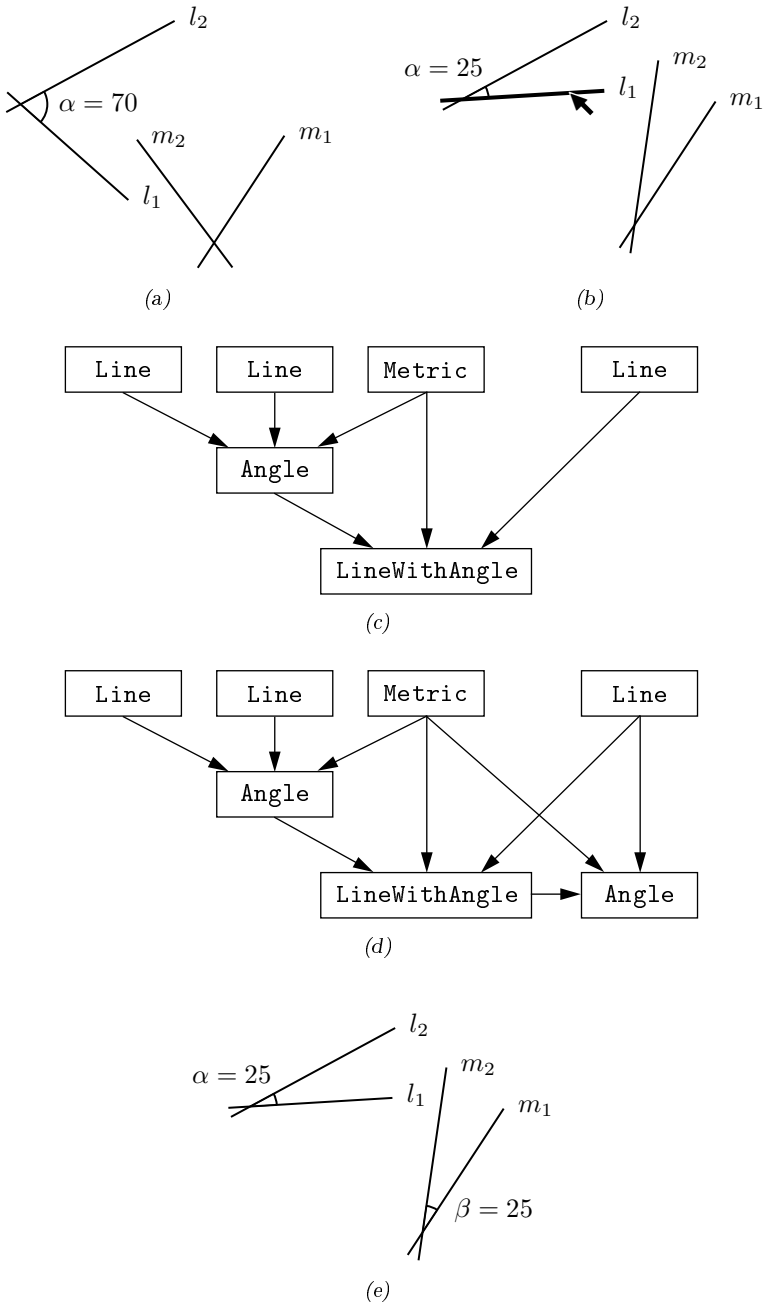
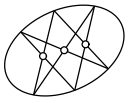


Figure 5.32. The orientation of m_2 is constrained by the angle between l_1 and l_2.



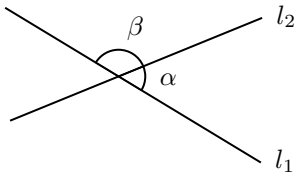


Figure 5.33. Complement angles.

- Is the angle directed (i.e. can it be negative) and if so, what is the "positive turn" in a general non-Euclidean metric?
- Angles may be complex and infinite in non-Euclidean geometries.

Since it must be possible to connect nodes like `Angle` and `LineWithAngle` to any type of metric node, we need a representation of angles and distances that works in all geometries and which does not suffer from the problems just mentioned. Let us see how such a representation can be found.

In Section 4.12 we saw that a metric can be defined in terms of a (possibly degenerated) conic and a constant which fixates the unit distance. The angle between two lines l_1 and l_2 which intersect in a point p was defined as

$$\operatorname{ang} l_1 l_2 = \frac{1}{2i} \ln (u\ v \mid l_1\ l_2) \tag{5.15}$$

where u and v are the ideal lines on p.

From Section 4.7 we know that if l_1 is distinct from u and v, there is a one-dimensional projective coordinate system on p in which u is the infinity point, v is the origin, and l_1 is the unit point. Let the homogeneous coordinates of l_2 in that system be

$$[l_2]_{u,v,l_1} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

Given another line m_1, a point q on m_1, and the coordinates $(c_1, c_2)^T$, we can unambiguously determine the position of a line m_2 through q such that

$$[m_2]_{u',v',m_1} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

where u' and v' are the ideal lines through q. Since

$$(u\ v \mid l_1\ l_2) = \frac{c_1}{c_2} = (u'\ v' \mid m_1\ m_2)$$

(Section 4.9), we see from Equation 5.15 that

$$\measuredangle l_1 l_2 = \measuredangle m_1 m_2$$

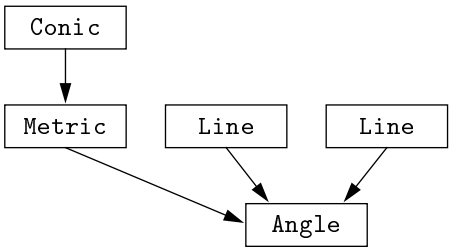


Figure 5.34. The parent of the `Metric` *node is the absolute conic.*

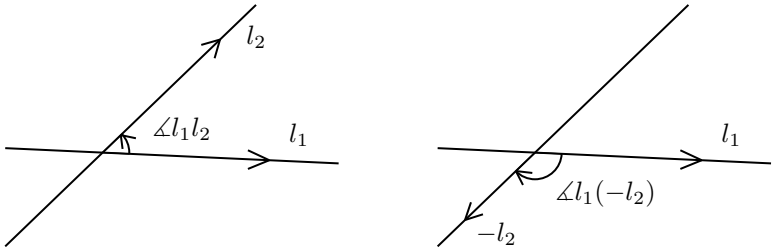


Figure 5.35. The angle between two directed lines.

Thus, we can use $[l_2]_{u,v,l_1}$ to represent the directed angle between l_1 and l_2. We will call $[l_2]_{u,v,l_1}$ the *relative orientation* of l_2 with respect to l_1 and denote it $\Theta(l_1, l_2)$.

The relative orientation does not suffer from the problems listed above:

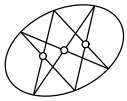
- The modulo π problem is caused by the logarithm in (5.15). For a complex number $z = re^{i\varphi}$, $\ln z = \ln|z| + i \arg z = \ln r + i(\varphi + 2\pi n)$, where n is an undetermined integer. If l_1 and l_2 are real and the angle is elliptic (Section 4.11) as in Euclidean geometry, then $u = \overline{v}$ and $|(u\ v \mid l_1\ l_2)| = 1$ (Section 4.9). Hence

$$\measuredangle l_1 l_2 = \frac{1}{2i} \ln (u\ v \mid l_1\ l_2) = \frac{1}{2i} \ln e^{i\varphi} = \frac{1}{2i} i(\varphi + 2\pi n) = \frac{\varphi}{2} + \pi n$$

  By avoiding the logarithm function and using the relative orientation $\Theta(l_1, l_2)$ in all calculations we get rid of the undetermined integer n.

- Complementary angles no longer cause ambiguities. There is a one-to-one correspondence between $\Theta(l_1, l_2)$ and the position of l_2 relative to l_1.

- If the order of the ideal lines u and v is reversed when the coordinate system on p is defined, the first and second coordinate of l_2 will be swapped, i.e., if $[l_2]_{u,v,l_1} = (c_1, c_2)^T$ then $[l_2]_{v,u,l_1} = (c_2, c_1)^T$. The corresponding cross-ratio is inverted: $(u\ v \mid l_1\ l_2) = 1/(v\ u \mid l_1\ l_2)$ and since $\ln(1/z) = -\ln z$, the sign of $\measuredangle l_1 l_2$ is reversed. Thus, the problem of defining a consistent turn is equivalent to the problem of distinguishing the ideal lines from each other.



In Euclidean geometry where the absolute elements are the two fixed and distinct points I and J, that is no problem. We just define u to be the ideal line on I and v to be the ideal line on J. In non-degenerate geometries, we can use the orientation of p and the orientation of the absolute conic to distinguish the ideal lines (Sections 4.13 and 5.2.5).

- In oriented projective geometry, the lines l_2 and $-l_2$ are different. It is then natural to distinguish $\measuredangle l_1 l_2$ from $\measuredangle l_1(-l_2)$, see Figure 5.35. The measure (5.15) is the same for both angles since $(uv|l_1(-l_2)) = (uv|l_1 l_2)$. In contrast, the relative orientations $\Theta(l_1, l_2) = [l_2]_{u,v,l_1} = (c_1, c_2)^T$ and $\Theta(l_1, -l_2) = [-l_2]_{u,v,l_1} = (-c_1, -c_2)^T$ are distinct.
- The angle measure (5.15) is undefined if the cross-ratio is infinite or zero. That happens when l_2 is an ideal line, i.e., when l_2 coincides with u or v. (If l_2 is real, this cannot happen in Euclidean or elliptic geometry since u and v are always complex. However, in hyperbolic geometry the ideal lines might be real.) When we use relative orientations, there is no problem. If $l_2 = u$, then $[l_2]_{u,v,l_1} = (1, 0)^T$, and if $l_2 = v$, then $[l_2]_{u,v,l_1} = (0, 1)^T$.

In order to support the calculation of relative orientations all `Metric` types in pdb are based on the notion of absolute elements and ideal lines and points. Either the explicit coordinates of the absolute elements are stored inside a `Metric` object, or the absolute elements are represented by separate objects that can be manipulated on the screen. For example, the `Metric` node in Figure 5.34 is using the conic represented by the `Conic` node as the absolute one. If the shape or position of this conic is changed, all angle and distance measurements depending on the metric must be updated and therefore, the `Conic` node is a parent of the `Metric` in the dependency graph. Using the setup in Figure 5.34, the user can investigate, for example, what happens if the absolute conic is flattened out until it approaches the infinity line of Euclidean geometry.

Distances are measured and constrained in a similar way. In Figure 5.36a the distance d between two points p_1 and p_2 have been measured. The point q_2 has been constrained to be at distance d along a given line m from a fixed point q_1 on m. The corresponding dependency graph is shown in Figure 5.36b, where the `Distance` node represents the distance d and the `PointOnLineAtDistance` node represents the constrained point q_2. As before, the `Metric` node defines how the distance is measured.

In hyperbolic and elliptic geometry, the distance between two points p_1 and p_2 on a line l can be written

$$k \ln (s\, t \mid p_1\, p_2) \tag{5.16}$$

where s and t are the ideal points on l (Section 4.11). The constant k determines the unit length and can be chosen so that the distance between two real, ordinary points is real. The three points s, t and p_1 define a coordinate system on l, and $[p_2]_{s,t,p_1} = (c_1, c_2)^T$ is the coordinates of p_2 in that system. $(c_1, c_2)^T$ represents the (signed) distance between p_1 and p_2 and we will therefore call this coordinate

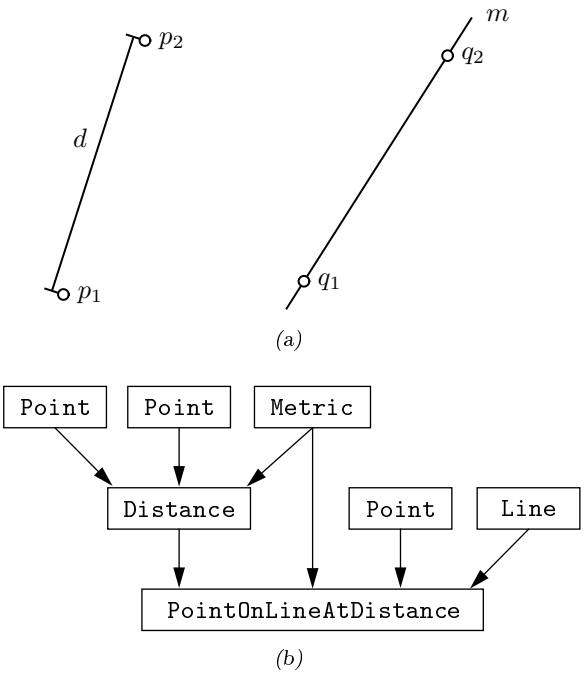


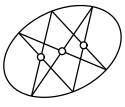
Figure 5.36. A constraint has been put on q_2 so that the distance q_1q_2 equals the distance p_1p_2.

the *relative position* of p_2 with respect to p_1 and denote it $\Delta(p_1, p_2)$. Thus, we can handle distances and angles in basically the same way. For the same reasons that we choose $\Theta(l_1, l_2)$ to represent angles, we prefer to use the relative position for representing distances rather than relying on the distance measure (5.16).

There is slight complication, though, with Euclidean distances. In Euclidean geometry, the ideal points s and t are the same since the absolute conic has degenerated into a double line, the line at infinity. Then s, t and p_1 no longer define a coordinate system on l and $(s\,t\,|\,p_1\,p_2)$ in the distance formula above will either be undefined or equal 1. Euclidean distances must therefore be calculated using the standard formula

$$\text{dist}\,(\begin{pmatrix}x_1\\y_1\\1\end{pmatrix},\begin{pmatrix}x_2\\y_2\\1\end{pmatrix}) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

(Section 4.11). However, in order to maintain a uniform representation of distances in the dependency graph, we must still be able to define a relative position $\Delta(p_1, p_2)$ for two Euclidean points p_1 and p_2. To do that, we obviously have to find another local coordinate system on l. The system must be chosen so that the coordinates are invariant under Euclidean isometries. More specifically, given an oriented Euclidean line l and a point p on l, we must define a one-dimensional



coordinate system $S(l,p)$ on l whose origin is p. Furthermore, S must have the following property: if q is a point on l and M is a orientation-preserving Euclidean isometry, then Mq should have the same coordinate in $S(M^{-T}l, Mp)$ as q has in $S(l,p)$. To achieve this, we can define S as follows. Let s be the infinity point on l and let r be a point at unit distance from p in the positive direction of l (i.e., at unit distance according to the Euclidean distance formula above) . Let $S(l,p)$ be the coordinate system on l in which p is the origin, s is the infinity point, and r is the unit point. To see that this definition works, let s' be the infinity point on $M^{-T}l$ and let r' be the point on $M^{-T}l$ at unit distance from Mp. Since M preserves the line at infinity, $s' = Ms$, and since M preserves distance and orientation, $r' = Mr$. Hence, $[Mq]_{s',p',r'} = [Mq]_{Ms,Mp,Mr} = [q]_{s,p,r}$.

The discussion above leads to the following definition of the interface to the `Metric` classes:

- Given a base line l_1 and another line l_2, return $\Theta(l_1, l_2)$, i.e., the relative orientation of l_2 with respect to l_1.
- Given a base point p_1 and another point p_2, return $\Delta(p_1, p_2)$, i.e., the relative position of p_2 with respect to p_1.
- Given a point p, a base line l_1 on p, and a relative orientation $(c_1, c_2)^T$, return the line l_2 on p for which $\Theta(l_1, l_2) = (c_1, c_2)^T$.
- Given a line l, a base point p_1 on l, and a relative position $(c_1, c_2)^T$, return the point p_2 on l for which $\Delta(p_1, p_2) = (c_1, c_2)^T$.
- Return the scalar angle value corresponding to a relative orientation $\Theta(l_1, l_2)$ (Equation 5.15).
- Return the scalar distance value corresponding to a relative position $\Delta(p_1, p_2)$ (Equation 5.16).

The last two functions in the `Metric` interface are only used for displaying angle or distance values on the screen, as shown in Figure 5.31a.

How many `Metric` nodes do we need in the dependency graph? Obviously, we will need one metric node for each type of geometry we want to work with. However, as we shall see in Section 5.3.3, there are situations where several metric nodes of the same type are required.

5.3 User interface design

5.3.1 A multi-view approach

Many word processors, drawing editors and CAD systems allow the user to create several *views* of a text or of a drawing. A view is simply a window through which a part of the document can be seen. A change to the underlying document is

immediately reflected in all open views. By creating multiple views, the user can look at different parts of the document at the same time.

Having multiple views is particularly important when using a dynamic geometry system, for several reasons. First, a single view cannot show the entire projective plane. In particular, for every (Cartesian) view there will be a corresponding line at infinity which will always be invisible in that view, no matter how large we make the viewport. However, if we have multiple views we can select the coordinate systems so that the line at infinity in one view becomes an ordinary line in another view. Second, there are many ways of visualizing the projective plane. For example, the base element "line" will be drawn as an ordinary, straight line in a Cartesian view, as a circle in the Poincaré disc model of the hyperbolic plane (Section 4.12.1), and as a box in a dependency graph. Multiple views allow the user to compare different representations of the same drawing. Third, different views can show different levels of detail, which is important when interacting with complicated drawings (Section 5.3.5).

There are currently three types of views available in pdb:

- `CartesianView`, in which the geometric base elements "points" and "lines" are drawn as ordinary points and straight lines.

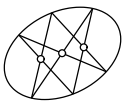
- `PoincareView`, in which the geometric "line" elements are drawn as Euclidean circles on a Poincaré disc (see Section 4.12.1).

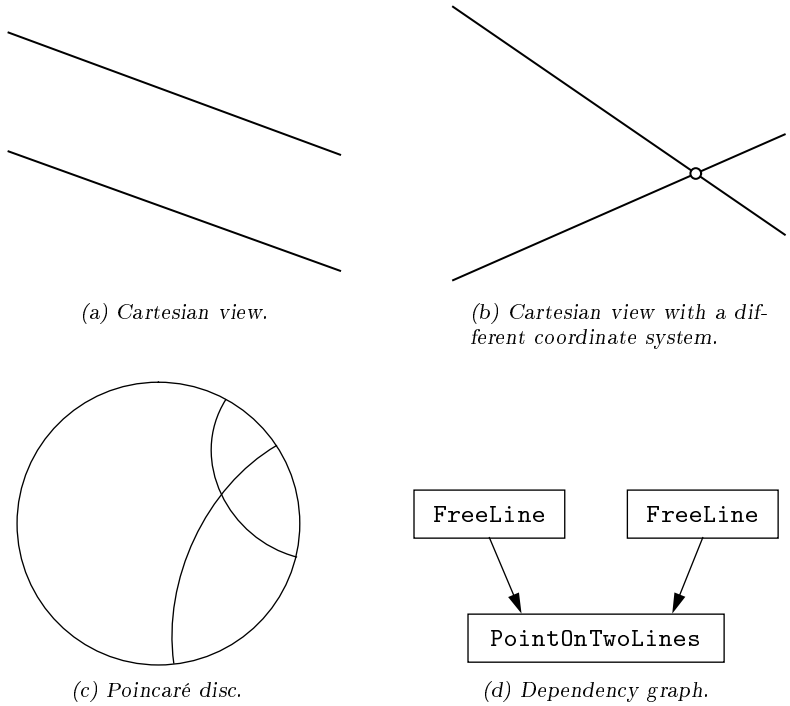
- `DependencyView`, in which points, lines and conics are represented by symbols (typically rectangular boxes with text labels), and dependencies are represented by arcs.

All views can be panned and zoomed individually. In addition, Cartesian views and Poincaré views can apply a general, user-defined projective transformation to all objects before displaying them.

In Figure 5.37, four views of the same drawing are shown. The drawing consists of two lines and their point of intersection. In the Cartesian view shown in Figure 5.37a, the lines are parallel, which means that their intersection point is at infinity. The view in Figure 5.37b is also Cartesian, but the coordinate system has been chosen so that the line at infinity in view (a) is an ordinary line here. The intersection point is therefore visible and the user can select it and interact with it. Figure 5.37c shows the corresponding Poincaré view. The last view is a `DependencyView` which shows the constructional history of the drawing. The intersection point is represented by a `PointOnTwoLines` node. We can see that its position is determined by the two unconstrained lines.

If a point, line or conic is assigned a name or a color by the user, the object will be consistently displayed with that name and color in all views. If an object is selected in one view, it will be highlighted in all views simultaneously. This makes it easy to identify corresponding objects in different views.





(a) Cartesian view.

(b) Cartesian view with a different coordinate system.

(c) Poincaré disc.

(d) Dependency graph.

Figure 5.37. Four views of the same drawing.

5.3.2 Modal and non-modal interaction

In most word processors and drawing editors, mouse interaction follows the *select-then-operate* model; the user first selects the operands (say, two lines), and then invokes an operation (for example, "create a point on two lines"). This interaction model is said to be *modeless* because the user does not have to put the program in a particular mode, for example by selecting a tool, in order to operate on an object. Any operation can be invoked at any time, provided that the correct number and type of operands have been selected. The select-then-operate interaction model is supported by pdb. In Figure 5.37a, we could have created the intersection point by selecting the two lines and choosing `Point on Two Lines` from the `Macro` menu.

However, the select-then-operate model has some shortcomings. First, the user must know which type of operands is required for a particular operation. That is no problem in an application such as a file manager where the objects on the screen belong to one or two different types (e.g. files and directories) and most operations are applicable to both types. But when most operations require different types of operands, the select-then-operate model becomes difficult to use. The system cannot help the user select an appropriate set of operands since it does not know what the user intends to do. Furthermore, the user often wants to

place new objects on the screen at the moment they are created. This applies in particular to drawing editors. The problem with the select-then-operate model is that the user creates a new object by selecting a menu item and therefore, the cursor provides no useful positioning information. The new object usually has to be placed in an arbitrary position by the system, and although the object can be dragged into the desired position, most users find that awkward. Also, in a complicated drawing it can be hard to see where the new objects go when they are created.

For this reason, pdb also supports *modal interaction*. The user can select one of several *tools* which determines the effect of subsequent mouse clicks and cursor movement. For example, to create a movable point on a line (a `PointOnOneLine`), the user would select the point tool and then click on the line to which the point should be attached. The system would then create the new point and place it on the line as close to the cursor as possible. As long as the point tool is selected, the user may create additional points by clicking the mouse button repeatedly. Modal interaction is therefore usually faster than the *select-then-operate* style of interaction.

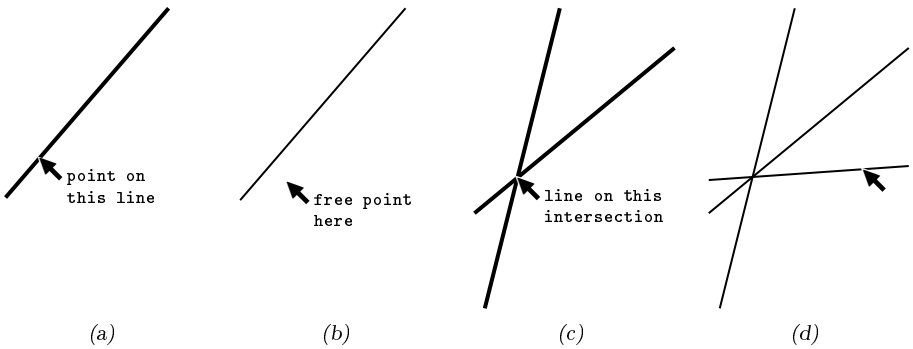
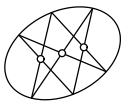


Figure 5.38. Feedback from the tool.

The type of object created by a certain tool depends on the type of objects under the cursor and, in some cases, the type of objects currently selected. For example, if the point tool is active and there is a line under the cursor, pdb will tell the user that it intends to create a `PointOnOneLine` if the mouse button is pressed, as shown in Figure 5.38a. On the other hand, if the cursor is over the window background, pdb will create a `FreePoint` at the cursor position (Figure 5.38b). If the mouse button is pressed while the cursor is over the intersection of a conic and a line, a `PointOnConicAndLine` will be created, etc.

A tool may create several objects in one operation. If we activate the line tool and point to the intersection of two lines, pdb will propose a line attached to the intersection point (Figure 5.38c). If the user accepts by clicking the mouse button, pdb will first create the intersection point (a `PointOnTwoLines`) and then create the line (a `LineOnOnePoint`), see Figure 5.38d.



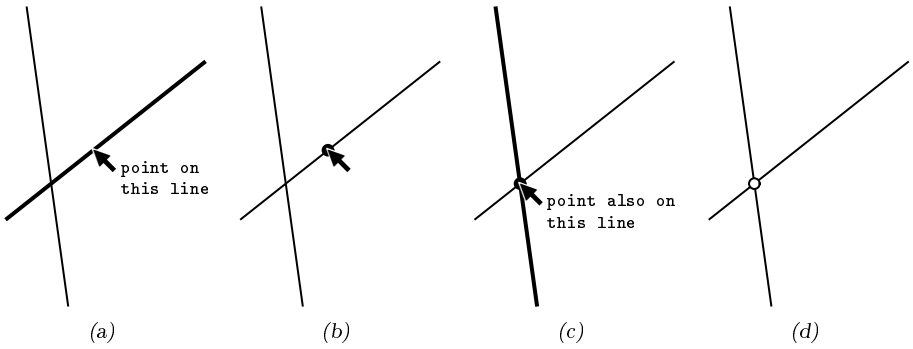


Figure 5.39. Another constraint can be added before the mouse button is released.

A newly created object can be dragged as long as the mouse button is held down. If the new object is dropped (i.e. the mouse button is released) over an existing object, a new constraint may be added to the new object. Figure 5.39a shows the feed-back provided by pdb when the point tool is active and the cursor is over a line. In Figure 5.39b, the user has pressed the mouse button and a `PointOnOneLine` has been created. Then, while holding the mouse button down, the user drags the point onto an intersecting line (Figure 5.39c). Finally, in Figure 5.39d, the point has been dropped onto the intersection and the point type has been transformed into a `PointOnTwoLines`.

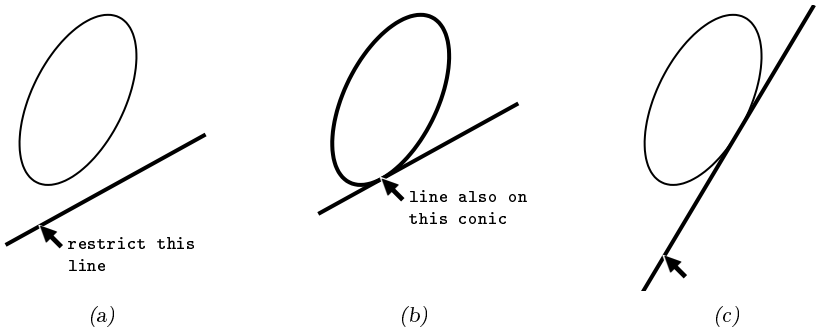


Figure 5.40. Constraints can be added to existing objects.

New constraints can also be placed on existing objects. In Figure 5.40a, a `FreeLine` is gripped with the selector tool or line tool active an with the `Control` key held down. In this case pdb proposes to restrict the position of the line. When the line is dropped onto the conic in Figure 5.40b, the line is transformed into a `LineOnOneConic` (Figure 5.40c).

Incidence constraints can be removed by "tearing" objects apart. Figure 5.41 shows a point p on a conic C, a free point q and a line l on p and q. If the line tool is active, the `Control` key is held down and the cursor is over p, pdb will propose

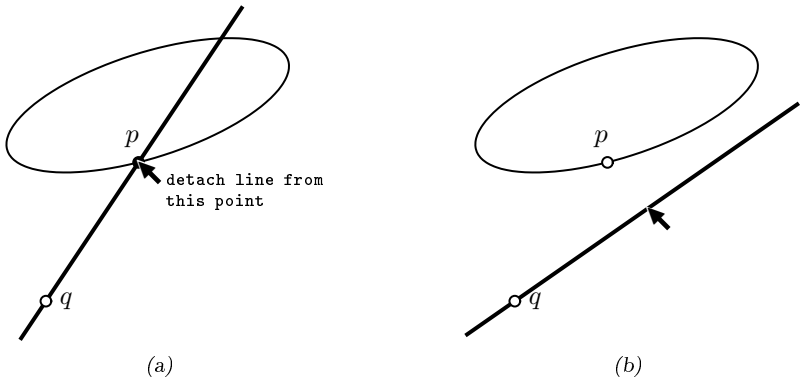


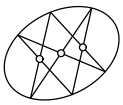
Figure 5.41. Constraints can be removed.

to detach the line from p. Note that the cursor must be over the incidence point; if the user just pointed somewhere on l, pdb would not know whether to detach l from p or from q. The line tool must also be active, otherwise pdb would not know whether the user wants to detach l from p or detach p from the conic.

In general, a tool tries to make use of as many of the objects under the cursor as possible. There are, however, exceptions to this rule. Figure 5.42a shows a conic C with a point p attached to it. If the line tool is active and the cursor is over p, the tool could create a line attached only to p (a `LineOnOnePoint`), a free tangent to C (a `LineOnOneConic`) or a line through p tangent to C (a `LineOnConicAndPoint`). If the tool would try to make use of as many objects as possible, it would choose the last alternative, a tangent through p, since that would involve both C and p. Unfortunately, that would prevent us from creating a line on p that is *not* tangent to C. For that reason, the line tool will choose the second alternative, a free line on p, in this situation. To create the tangent line through p, we can simply press the mouse button over p, move the cursor to the conic and release the button (Figures 5.42b-c). While the button is pressed, the line is a `LineOnOnePoint`. When the button is released, and additional constraint is added and the line is converted to a `LineOnConicAndPoint`. If we instead want a tangent to C not attached to p, we can click anywhere on C *except* on p (Figures 5.42d-e).

No tool will allow the user to add constraints that would create cycles in the dependency graph. Figure 5.43a shows a simple drawing consisting of three lines l_1, l_2, l_3 and three points p_1, p_2, p_3. The corresponding dependency view is shown in Figure 5.43b. l_1 is a free line, p_1 is a point on l_1, l_2 is a line on p_1, p_2 is a point on l_2, l_3 is a line on p_2, and finally, p_3 is a point on l_3. If we were allowed to drag l_1 and drop it onto p_3, we would get the dependency graph shown in Figure 5.43c. Since the graph contains a cycle, this drag-and-drop operation is disallowed.

There is one tool for each type of primitive geometric object: point, line, and conic. There is also a measurement tool for measuring distances and angles, and a selector tool which is used for dragging, selecting and deselecting objects. In



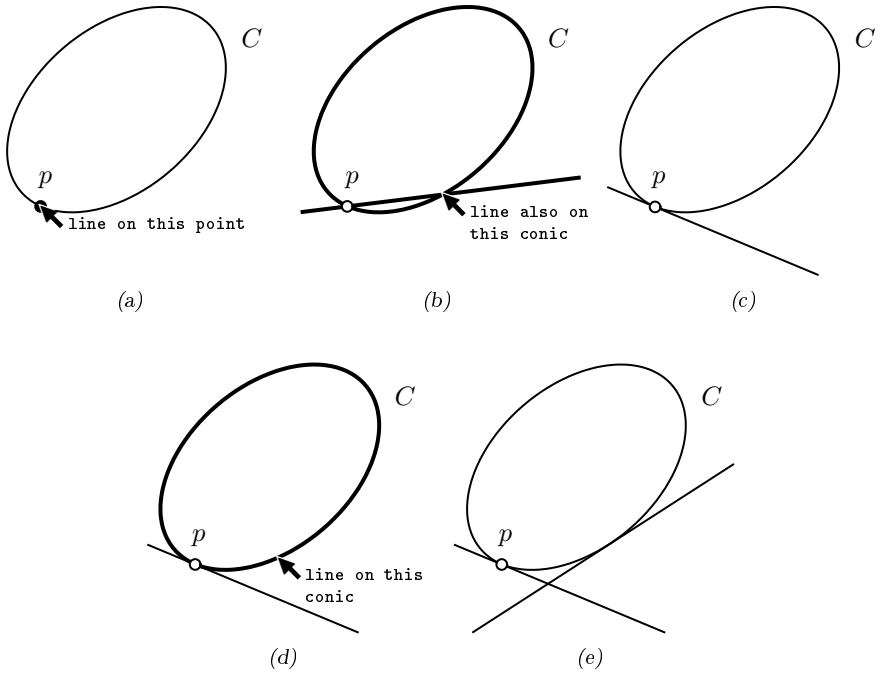


Figure 5.42. Creating a line on p, a tangent through p and a tangent not on p.

Section 5.4.6 we will discuss the implementation of the tools and how new tools can be added.

Drag-and-drop operations work smoothly only if all operands are visible in the same view. If we would create, say, a line on a point in one view and drag it to a second point in another view which has a different coordinate system, the line would have to jump at some moment during the dragging operation since the orientation and position of the line will not be the same in both views. In situations like that, the *select-then-operate* interaction model is easier to use; we could simply select the two points, one in each view, and apply `Macro→Line on Two Points`.

User-defined macros (Section 5.3.6) fetch their operands from the current selection when they are invoked. Thus they only support the select-then-operate interaction style. For example, to invoke the `Focal Points` macro, the user first selects a conic, the chooses `Macro→Focal Points of Conic`. Unless there is exactly one conic currently selected, an error message will be generated. Alternatively, the system could turn macros into user-defined tools which would ask the user to select suitable operands. For example, the focal points macro could ask the user to select a conic. However, unlike the built-in tools, such "macro tools" would not be able to suggest different operations depending on what the user is pointing to and therefore their usefulness would be limited. Creating "intelligent"

tools requires a significant amount of programming.

5.3.3 Using metric information

When we described the basic interaction model in the previous section, we concentrated on *incidence* constraints, because they are particularly easy to define and redefine using a drag-and-drop interface. When we deal with *metric* information, such as distances and angles, things get more complicated. The user should be able both to *measure* distances and angles and to *place constraints* on them interactively. In particular, it should be easy to copy a distance or angle from one part of a drawing to another, i.e. to say "make this angle equal to that one". When a metric constraint is specified, it must be made clear which of the objects involved the user wants to update in order to satisfy the constraint.

The dependency graph nodes needed to represent measurements and metric constraints were discussed in Section 5.2.7. In this section, we will concentrate on the user interaction aspects.

Figure 5.31 on page 125 showed how an angle is displayed in an Cartesian view and how it is represented in the dependency graph. But how was the `Angle` node created in the first place? The simplest way to define an angle is to use the measurement tool, as shown in Figure 5.44. With the measurement tool active and the cursor over l_1, we get the feedback shown in Figure 5.44a. We accept by pressing the mouse button, and with the mouse button down we move the

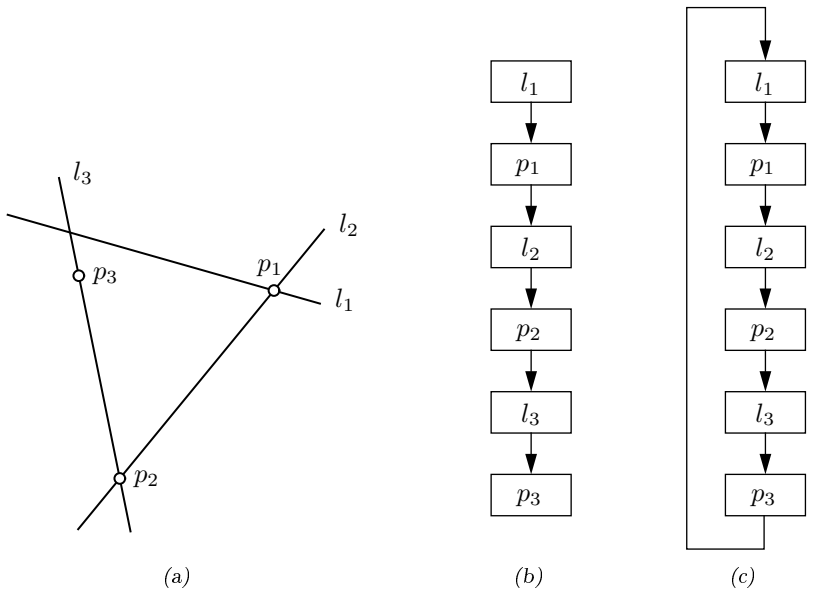
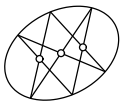


Figure 5.43. If the user were allowed to attach l_1 to p_3, a cycle would be created in the dependency graph.



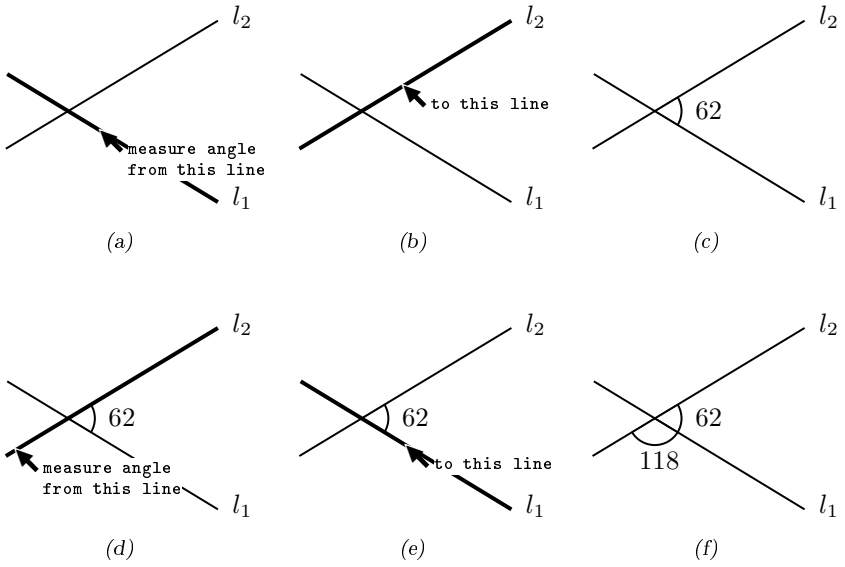


Figure 5.44. Measuring angles.

cursor to l_2 (Figure 5.44b) and release the button (Figure 5.44c). It is at this moment that the `Angle` node in Figure 5.31b is created. The small circular arc and the angle value are placed close to the intersection point. It is important on which side of the intersection point the mouse button is pressed and released. Figures 5.44d-f shows how to measure the complement angle.

In Figure 5.32 on page 126, a constraint was placed on the position of a line m_2 making $\measuredangle l_1 l_2 = \measuredangle m_1 m_2$. Thus, the position of m_2 was determined by the three other lines. Such a constraint can be created in two steps, starting from two pairs of free lines, as shown in Figure 5.45. First, the angles $\measuredangle l_1 l_2$ and $\measuredangle m_1 m_2$ are measured using the measurement tool (Figures 5.45a-f). Then, with the measurement tool still active, the arc representing $\measuredangle l_1 l_2$ is dragged onto the arc representing $\measuredangle m_1 m_2$ (Figures 5.45f-h). When the mouse button is released in Figure 5.45g, the measurement tool converts the `FreeLine` node representing m_2 to a `LineWithAngle` node. Then, if any of the lines l_1, l_2 or m_1 is dragged, m_2 will be updated (Figures 5.45i-j).

If m_2 had been a `LineOnOnePoint`, then it would have been converted to a `LineOnPointWithAngle` instead. However, if it had been a `LineOnTwoPoints`, the position of the line would already have been completely determined and therefore, the arc representing $\measuredangle m_1 m_2$ would not have been an acceptable drop target.

Note that the drag-and-drop operation in Figures 5.45f-g restricted the position of m_2, not that of m_1. This was due to the way $\measuredangle m_1 m_2$ was defined in Figures 5.45c-d. Because m_2 was selected after m_1, the subsequent drag-and-drop operation modified the constraints of m_2. In fact, identifying the target node is

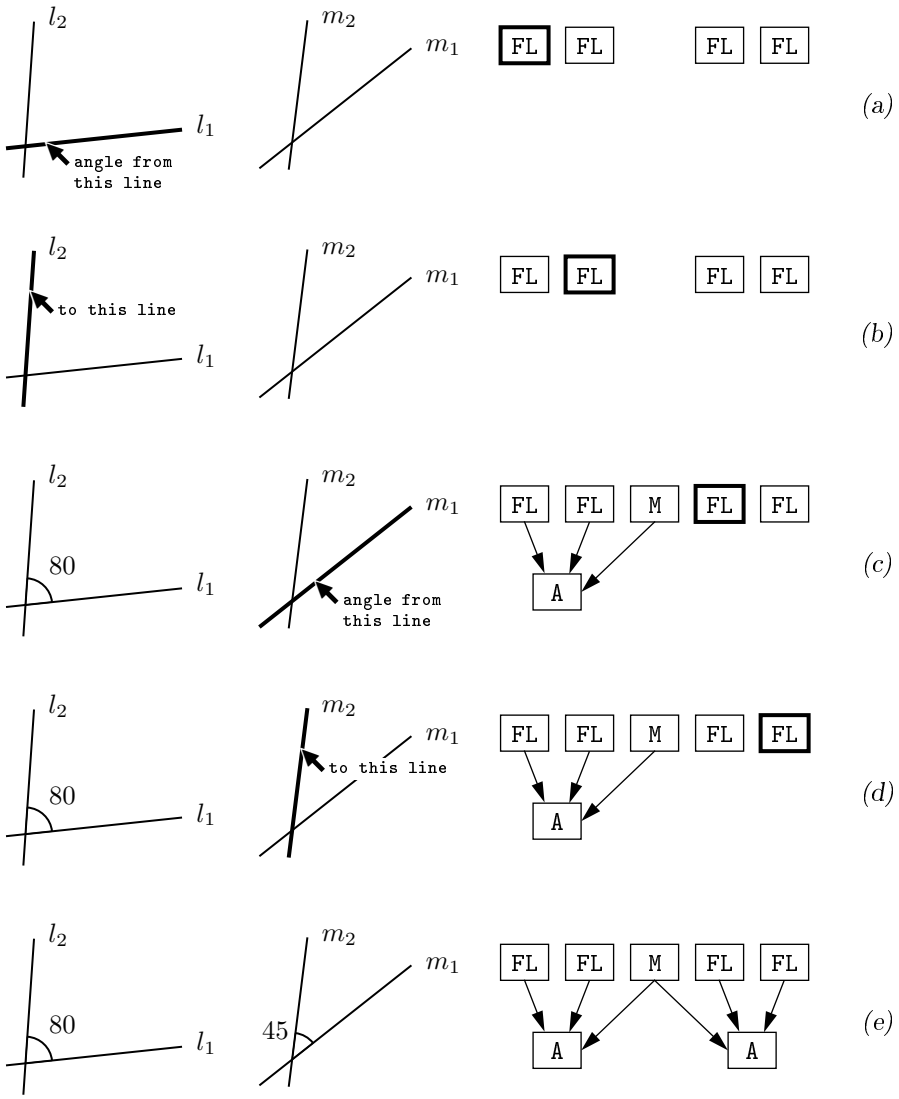
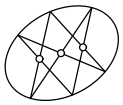


Figure 5.45. Copying an angle. Legend: FL=FreeLine, M=Metric, A=Angle, LWA=LineWithAngle.



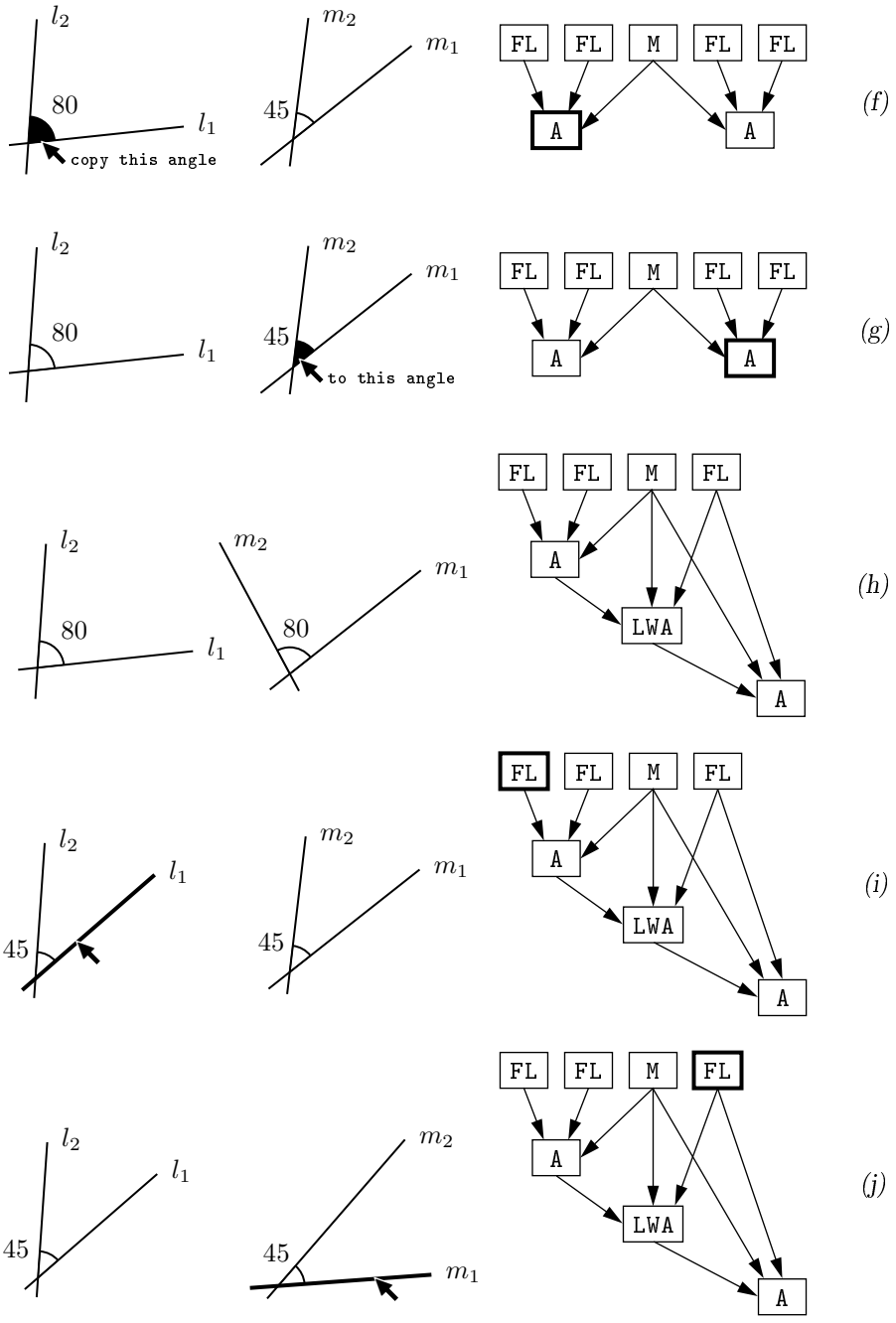


Figure 5.45. Continued.

the main reason why constraints must be specified by dragging angles onto angles. For example, if we were allowed to drop $\measuredangle l_1 l_2$ directly onto the intersection of m_1 and m_2, it would not be clear whether we wanted to restrict the position of m_1 or m_2. Furthermore, it would not be clear which of the two complementary angles we wanted to equal $\measuredangle l_1 l_2$. Constraints involving distances are defined in a similar manner.

So far, we have not mentioned where the `Metric` nodes in the dependency graph come from. In general, when we measure distances and angles in a Cartesian view (Section 5.3.1) we want the numerical values to match what we actually see on the screen. If an angle between two lines appears to be 45 degrees, the displayed angle value should read `45`. If the distance between two points is 5.5 cm on the screen, the distance value should read `5.5`. To accomplish that, the system must choose a Euclidean metric which matches the view we are looking at. The infinity line of the metric has to coincide with the infinity line of the view, and the unit distance defined by the metric much match the unit distance of the screen.

However, the situation is complicated by the fact that the user is allowed to create more than one view. The view coordinate systems are related by projectivities that do not have to be Euclidean isometries (Section 5.3.1). Consequently, an angle or a distance will not be the same in all views. Furthermore, a Poincaré view is usually associated with an *hyperbolic* metric whose absolute conic coincides with the circumference of the Poincaré disc.

Thus, each view is associated with a different metric, and it will not be possible to associate a given `Angle` or `Distance` node with a single numerical value. For this reason, `Angle` and `Distance` nodes have been designed not to compute a value until another object asks for it. At that point, the requesting object will specify the metric in which to measure the angle or distance. In this way, a view can obtain and display the angle and distance values that correspond to the intrinsic metric of that particular view. Hence, the dependency graph in Figure 5.45 is obviously simplified. The `Angle` node has in fact no single, fixed `Metric` parent node. Instead, the metric is provided by the context.

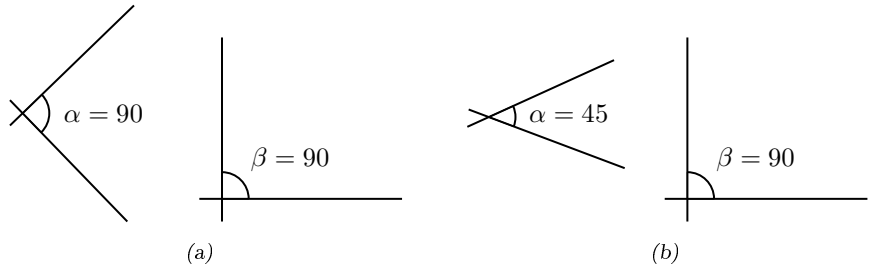
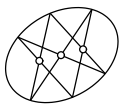


Figure 5.46. Two different projections of the same angles.

The fact that different views are associated with different metrics is not a big problem as long as we only perform *measurements*. The system just has to make sure that the numerical values displayed in each view have been computed in the



appropriate metric. However, when we define metric *constraints*, we must choose a specific metric. For example, two angles that are equal in one metric might not be equal in another. To see that, consider Figures 5.46a and 5.46b which show two different views of a drawing which consists of four lines. The views are related by a projectivity T which rotates the Euclidean plane in R^3 around the horizontal x-axis[4]. Evidently, α is affected by T while β is not. This shows that it is not possible to establish a simple relationship between angles and distances measured in (a) and angles and distances measured in (b), even if we know the projectivity T. We cannot, for example, say that an angle of 90 degrees in (a) corresponds to an angle of 45 degrees in (b). The relation between the angles in the two views is determined not only by T, but also by the position and orientation of the lines involved. Therefore, a metric constraint will in general hold in only one view. For example, we may require that $\alpha = \beta$ in Figure 5.46a in which case $\alpha \neq \beta$ in (b), or we may require $\alpha = \beta$ in (b) in which case $\alpha \neq \beta$ in (a).

By default, the system makes sure that a constraint is satisfied in the view in which it was defined. For example, if the drag-and-drop operation in Figures 5.45f-h was carried out in a view V, the `Metric` parent of the resulting `LineWithAngle` node will be the metric associated with V. When the position of the line needs to be updated, the `LineWithAngle` will ask the `Angle` node for an angle value computed in that particular metric.

Although there is a natural metric associated with every view, the user can override it. Actually, different types of metrics and views can be combined freely. That is emphasized by the fact that the basic type of view, which has straight, orthogonal x and y axes, is called Cartesian, not Euclidean. It is perfectly possible to study hyperbolic geometry by combining a Cartesian view with a hyperbolic metric. On example of this is given in Section 6.4.1.

5.3.4 Visual representation of objects with complex coordinates

Consider again the construction of the polar line shown in Figure 5.1, page 81. As described in Section 4.8, the tangent lines will become conjugate complex if the point p is in the interior of the ellipse, provided that the ellipse is defined by an equation with real coefficients (i.e., its coefficient matrix C is real). The tangent points q_1 and q_2 will also become conjugate complex but the polar line, $q_1 \times q_2$, will remain real.

To our knowledge, no other dynamic geometry system can handle complex solutions to these equations. Usually, a polar line constructed in this way simply disappears when the point p is dragged into the ellipse. In contrast, pdb can not only perform complex arithmetic, but also visualize points and line whose coordinates are complex. Figure 5.47 shows what happens in pdb when p is dragged into the ellipse. In (c) the two tangent lines t_1 and t_2 and the two tangent points q_1 and q_2 are gray[5], which indicates that their coordinates have

[4] We are here referring to the standard R^3 embedding discussed in Section 4.1.

[5] On a color screen, they would typically be drawn in red.

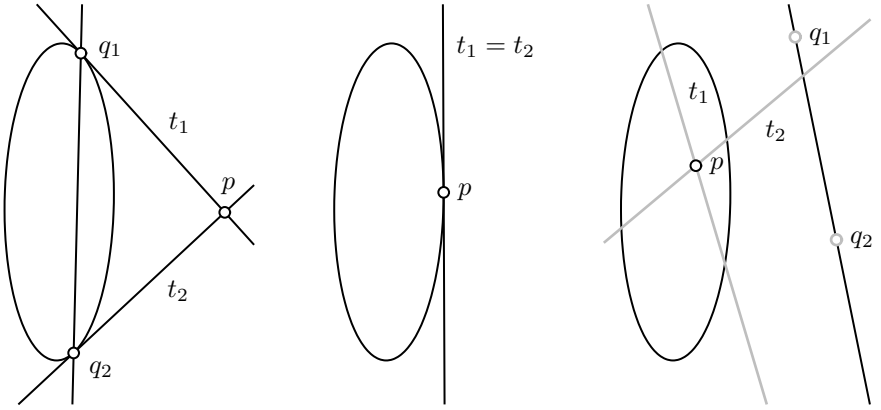


Figure 5.47. The polar of a point p with respect to a conic.

become complex. However, the line between the images of q_1 and q_2 is still black, which shows that the polar line is real.

Note that the tangents t_1 and t_2 in Figure 5.47 are at all times incident both with the conic and with the point p. However, when the coordinates of the tangents become complex, it is not possible to faithfully visualize these incidences on paper or on a computer screen; the lines are part of the complex projective plane which has four real dimensions, while a sheet of paper only has two. One can imagine that t_1 and t_2 in Figure 5.47c are floating in front of or behind the paper and that the gray lines is the result of a projection from the complex projective plane onto the real one.

We will refer to the gray lines in Figure 5.47c as the *real images* of the corresponding complex lines. The homogeneous coordinates of the real image of a complex point or line $a + ib,\ a, b \in \mathrm{R}^3$ are given by

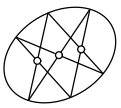
$$f(a + ib) = a + kb$$

where k is a real number. Why choose this particular function f? First, f should be the identity function if the argument is real. We can easily see that if $b = 0$, then $f(a + ib) = f(a) = a$. Second, f should be continuous so that the images do not jump if the underlying, complex points and lines are moved slightly. Again, it is evident from the definition above that f has this property.

We would also like f to preserve as many *incidence relationships* as possible. For example, if p is a complex point and l a complex line, it would be nice if the real image of p would be incident with the real image of l if and only if p were incident with l, i.e.,

$$f(p)^T f(l) = 0 \Leftrightarrow p^T l = 0$$

The following argument shows that it is not possible to find such a function. Let p and q be two arbitrary but distinct complex points. Choose any line l on p which



is not on q. If the equivalence above holds, $f(p)^T f(l) = 0$ and $f(q)^T f(l) \neq 0$, hence $f(p) \neq f(q)$. Therefore, $p \neq q \Rightarrow f(p) \neq f(q)$ and f must be 1-1. On the other hand, $f(f(p)) = f(p)$ since $f(p)$ is real and f is required to be the identity function for real vectors. Since f is 1-1, $f(p) = p$. But that is not possible since p was assumed to be complex and $f(p)$ must be real.

Nevertheless, we can require that a *real* point on l should be incident with the real image of l. That is, if p is real and $p^T l = 0$ then we require that $p^T f(l) = 0$. As we saw in Section 4.14, a complex line $l = u + iv,\ u, v \in \mathrm{R}^3$, only has one real point, $u \times v$. Let p be that point. When applying f to both p and l, we get

$$f(p)^T f(u + iv) = p^T(u + kv) = p^T u + kp^T v = (u \times v)^T u + k(u \times v)^T v = 0$$

Thus, the image of p is incident with the real image of l. Because of duality, a real line on a complex point p will be incident with the real image of p. In Figure 5.47c, for example, the image of t_1 is *not* incident with the image of q_1 since both t_1 and q_1 are complex. However, p is incident with the images of t_1 and t_2, and the polar line is incident with the images of q_1 and q_2 since both p and the polar line are real.

However, the mapping f cannot preserve the incidence relationship between a complex point $p = a + ib$ and a real conic. If p is on the conic, we have

$$p^T C p = (a + ib)^T C(a + ib) = (a^T C a - b^T C b) + i(a^T C b + b^T C a) = 0$$

and thus

$$a^T C a - b^T C b = 0 \tag{5.17}$$

$$a^T C b + b^T C a = 0 \tag{5.18}$$

If we insert $f(p)$ into the same equation we get

$$f(p)^T C f(p) = (a + kb)^T C(a + kb) = 0 \tag{5.19}$$

If, for simplicity, we choose $k = 1$ and use eq. 5.17 and 5.18 , the left-hand side of eq. 5.19 can be written

$$f(p)^T C f(p) = a^T C a + b^T C b + b^T C a + a^T C b = 2a^T C a$$

which will not be zero in general. The same applies to complex lines and real conics. This is of course obvious from the drawing in Figure 5.47c. If p is inside the ellipse, there is no way that the real images of t_1 and t_2 can be drawn as tangents to the ellipse and at the same time intersect in p.

Interestingly, the function f, as defined above, is invariant under real projectivities [Klein25]. If $a, b \in \mathrm{R}^3$, $T \in \mathrm{R}^{3\times 3}$, then

$$f(T(a + ib)) = f(Ta + iTb) = Ta + kTb = T(a + kb) = T(f(a + ib))$$

A slight problem with the definition of f above is that it is sensitive to the scaling of its argument. If s is a complex scalar, p and sp represent the same point

projectively. However, there is in general no real scalar r such that $f(sp) = rf(p)$. Thus, if p is scaled by a complex scalar, its real image will move. In practice, we can prevent that by scaling p so that its homogeneous coordinate becomes 1 before applying f. That will of course not be possible for points at infinity, but such points will not be visible on the screen anyway.

When we draw a conic, we usually draw the real solutions to the point equation $p^TCp = 0$. However, as we saw in Section 4.14, the point set of some conics contains no real points or only a few real points. How should these conics be displayed? One possibility is to take each complex point $p = a+ib$ on C and simply draw $f(p)$. But will the resulting point set resemble a conic on the screen, i.e., is there a real conic C' such that $f(p)^TC'f(p) = 0 \Leftrightarrow p^TCp = 0$? Unfortunately, the answer is no. Let C be the imaginary unit circle $x^2 + y^2 + 1 = 0$ (Section 4.14), which has no real points. Then C is the identity matrix and the homogeneous point equation can be written $p^Tp = 0$. With $p = a + ib$ we get

$$p^Tp = (a + ib)^T(a + ib) = a^Ta - b^Tb + i(a^Tb + b^Ta) = 0$$

a^Tb is a scalar and thus $a^Tb = (a^Tb)^T = b^Ta$. Furthermore, $a^Ta = |a|^2$ and $b^Tb = |b|^2$. Since both the real and imaginary part of p^Tp must be zero

$$|a| = |b|$$
$$a^Tb = 0$$

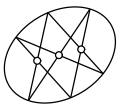
Any complex point $a + ib$, where the R^3 vectors a and b are perpendicular and have equal length, is on the conic. For every real vector q, we can find two perpendicular vectors a and b of equal length such that $q = a + kb$, where k is a real constant. In other words, for every real vector q there is a complex point $p = a + ib$ on C such that $f(p) = q$, $|a| = |b|$ and $a^Tb = 0$. Therefore, the set of real images of points on C, $\{f(p) : p^Tp = 0\}$, contains every real vector and cannot be the point set of any real conic.

However, we can define a real image of a conic that has at least one real point. Guided by the discussion of complex lines above, we say that $f(C) = A+kB$, $k \in \mathrm{R}$ is the real image of a complex conic $A + iB$, where A and B are real matrices. This mapping has some of the properties that the corresponding line mapping has. In particular, if p is a real point on C, that is, if $p^TCp = 0$, then p will be incident with $f(C)$:

$$p^TCp = p^T(A + iB)p = 0 \Rightarrow p^TAp = 0, p^TBp = 0 \Rightarrow p^T(A + kB)p = 0$$

since p, A and B are real. What does this mean? Assume that the user has created a conic C on five real points. If one of the points is, say, the intersection between a line and another conic, it can become complex at any time. If that happens, C will also become complex. However, its real image $f(C)$ will still pass through the four remaining real points on the screen.

If C has at least one real point, so does $f(C)$, and since $f(C)$ is real, it has infinitely many real points. Therefore, $f(C)$ can be drawn as a ordinary, real



conic. However, $f(C)$ can be degenerated even though C is proper. Actually, $f(C) = A + kB$ is a linear combination of two real matrices, both of which may be degenerated. Since C is a complex, A and B are linearly independent (Section 4.14.2). Hence, $A + kB$ represents a pencil of real conics, and every member of that pencil passes through the four (possibly complex) points where A intersects B. If C is proper, we can *choose* a k such that $f(C)$ is a proper real conic. Conversely, if C is degenerated, we can choose a k which makes $A + kB$ degenerated.

Since objects with real and complex coordinates are displayed differently, the system must check whether the coordinates of an object are real or not before drawing it on the screen. Because of round-off errors, it is not meaningful to test whether the imaginary part is exactly zero. Instead, pdb uses simple thresholding. A complex number $a + ib$, where $a, b \in \mathrm{R}$, is considered real if

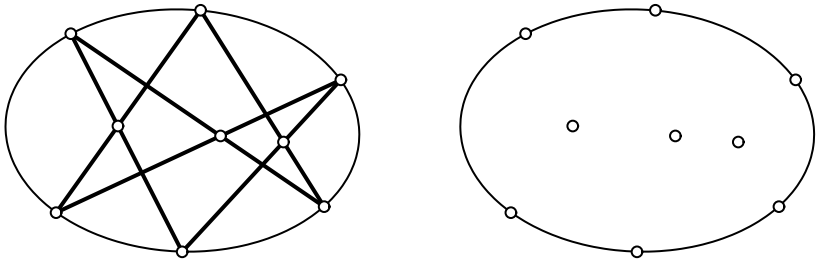
$$|\frac{b}{a}| < \theta$$

where θ is a small real constant. A numerical test is of course unreliable. If θ is too small, a coordinate which theoretically is real might be classified as complex. Actually, that could happen even with a large threshold because round-off errors can accumulate in complicated drawings. The outcome of the test will not affect the position of the object on the screen. Since f is continuous and has no effect on real vectors, we can safely apply it to the coordinates of every object. However, objects with complex coordinates are displayed in a different style and/or in a different color to show that they are not in the real plane. If a numerical test is used, small round-off errors may cause the color of the object to flicker, which can be very annoying. The only completely satisfactory solution is to use symbolic methods for determining whether the coordinates of an object are real or not. A number of proof engines that might be useful in this context are presented in [Kutzler86, Wu86, Richter-Gebert95].

5.3.5 Working with complicated drawings

One thing that a user of a dynamic geometry system will soon discover is that drawings quickly become very complicated and may very well contain more than a hundred objects. Such drawings are hard to interact with because of the clutter they generate on the screen. Text labels may help to identify the objects, but too many labels just make the clutter worse. Apart from the problem of telling which object is which, it is difficult to pick an object if it is close to or obscured by other objects. The user interface of pdb has several features which help the user to handle complex drawings.

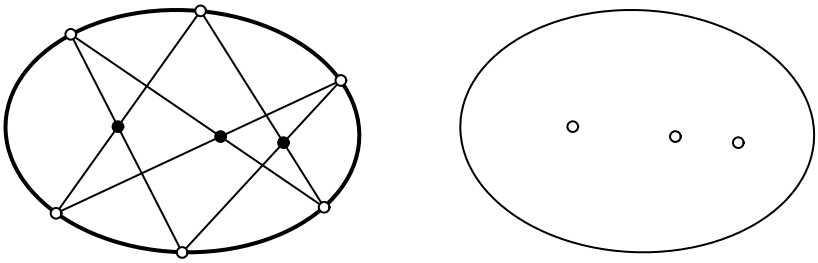
First, the objects in a view have a specific stacking order. Lower-dimensional objects, such as points, are always on top of higher-dimensional objects such as lines and conics. That makes it possible to pick points that are close to lines. When several points are displayed on top of each other, the user can change the stacking order.



(a) (b)

Figure 5.48. The images of six lines are selected and removed from the view. The line objects themselves are not destroyed.

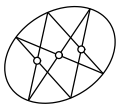
Second, a view may show a subset of the objects in a drawing. If a view shows too much detail, the user can either hide some of the objects or create a new view which shows fewer objects. In Figure 5.48a, which illustrates Pascal's theorem, the user has just selected the six auxiliary lines and is about to apply `Edit→Delete Image`. In Figure 5.48b the line images have disappeared, but the images of the intersection points remain. Removing the image of a shape in a view will not delete the underlying object in the drawing. Actually, a node in the dependency graph will not be deleted as long as some other node depends on it (directly or indirectly), or there exists an image of it in some view. In Figure 5.49a, the user instead selects the conic and the three intersection points, and then applies `File→New Cartesian View`. A second view is created, in which the images of the selected objects are shown (Figure 5.49b).



(a) (b)

Figure 5.49. The conic and three of the points are selected. A second view which shows only these objects has been created in (b).

Third, as we mentioned in Section 5.3.2, the currently active tool continuously monitors what is under the cursor and only highlights objects that it can act upon. Other objects are ignored. For example, in Figure 5.50 a label and an arc which represent an angle are partly obscured by several lines. This makes it difficult to



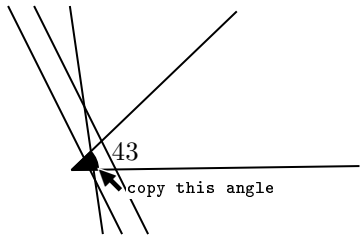


Figure 5.50. If the measurement tool is active, the lines obscuring the angle symbol will be ignored which makes it easy to select the angle.

select the angle for an operation. However, if the measurement tool is activated, the angle will always be given priority, which is indicated by cursor text.

Finally, dependency views make it easy to select the right objects. Figure 5.51a shows a conic defined by five points. An additional point has then been attached to the conic. Suppose that we want to create a tangent to the conic through this sixth point. How do we find it? We could drag each of the points in turn, and look for the one that does not affect the shape of the conic. However, it is much easier to bring up the dependency view shown in Figure 5.51b. If we click on the point that depends on the conic, the corresponding point will be highlighted in the Cartesian view, and we can define the tangent through it (Figure 5.51c). Alternatively, we could select the conic and the sixth point in the dependency view and directly select the `Macro→Line on Conic and Point` menu item.

5.3.6 Macros

pdb's built-in tools make it easy to perform simple operations, such as creating a single object or adding an incidence constraint. The tools were designed to make that kind of interaction as smooth and intuitive as possible. However, creating large drawings with dozens of objects using only the built-in tools can be tedious. The user will find himself making the same basic constructions, such as angle bisectors and polar lines, over and over again. To alleviate that problem, the user can define *macros* for often-needed constructions. pdb macros are written in Tcl (Tool Command Language) [Ousterhout94]. The Tcl interpreter has been extended with functions that gives the user full access to all the internal data structures in pdb. The choice of macro language and its integration with the pdb kernel will be discussed in Section 5.4.3. Here, we will describe how macros are defined and invoked by the user.

The easiest way to create a macro is to save a part of an existing drawing as a macro. pdb then generates Tcl code which will repeat the construction each time it is executed. If necessary, the Tcl code can be modified in a text editor. The macro can also be written from scratch.

Consider the polar line construction illustrated in Figures 5.52a-h. To save this construction as a macro, we select the *arguments* (or input objects) of the macro, in this case the given point and the conic, and invoke `File→Create`

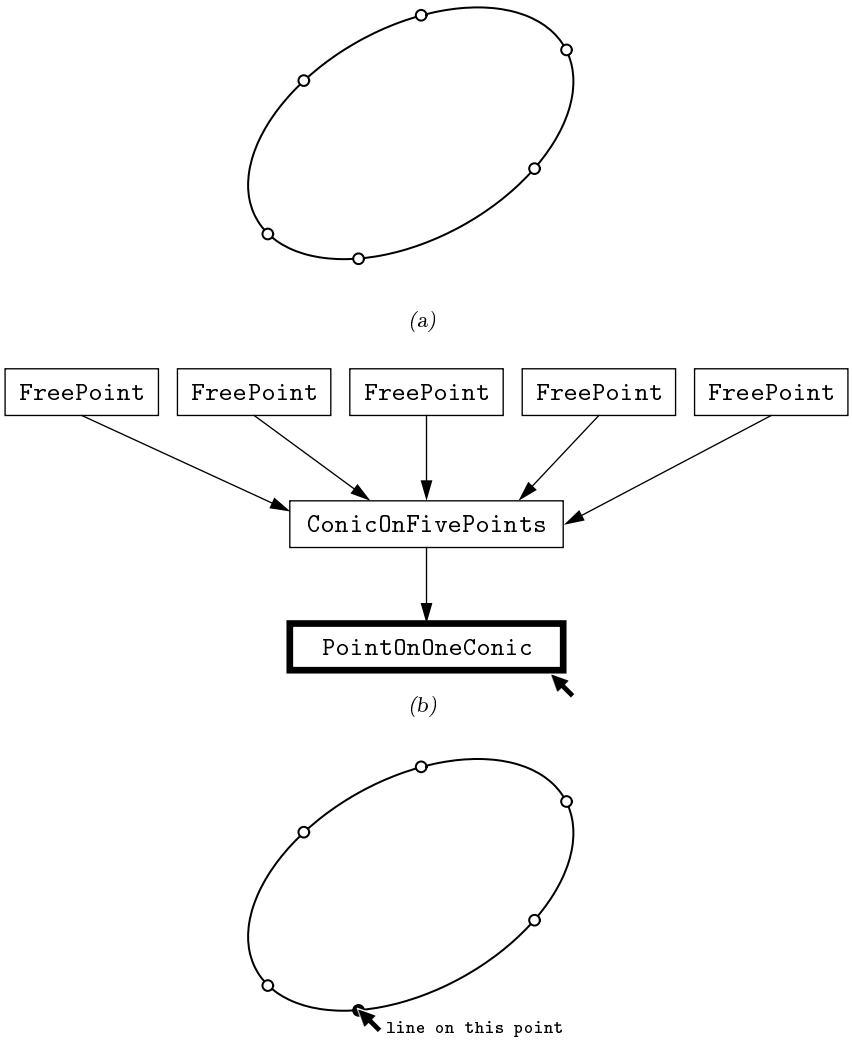
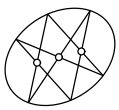


Figure 5.51. Sometimes it easier to choose the operand in a dependency view.



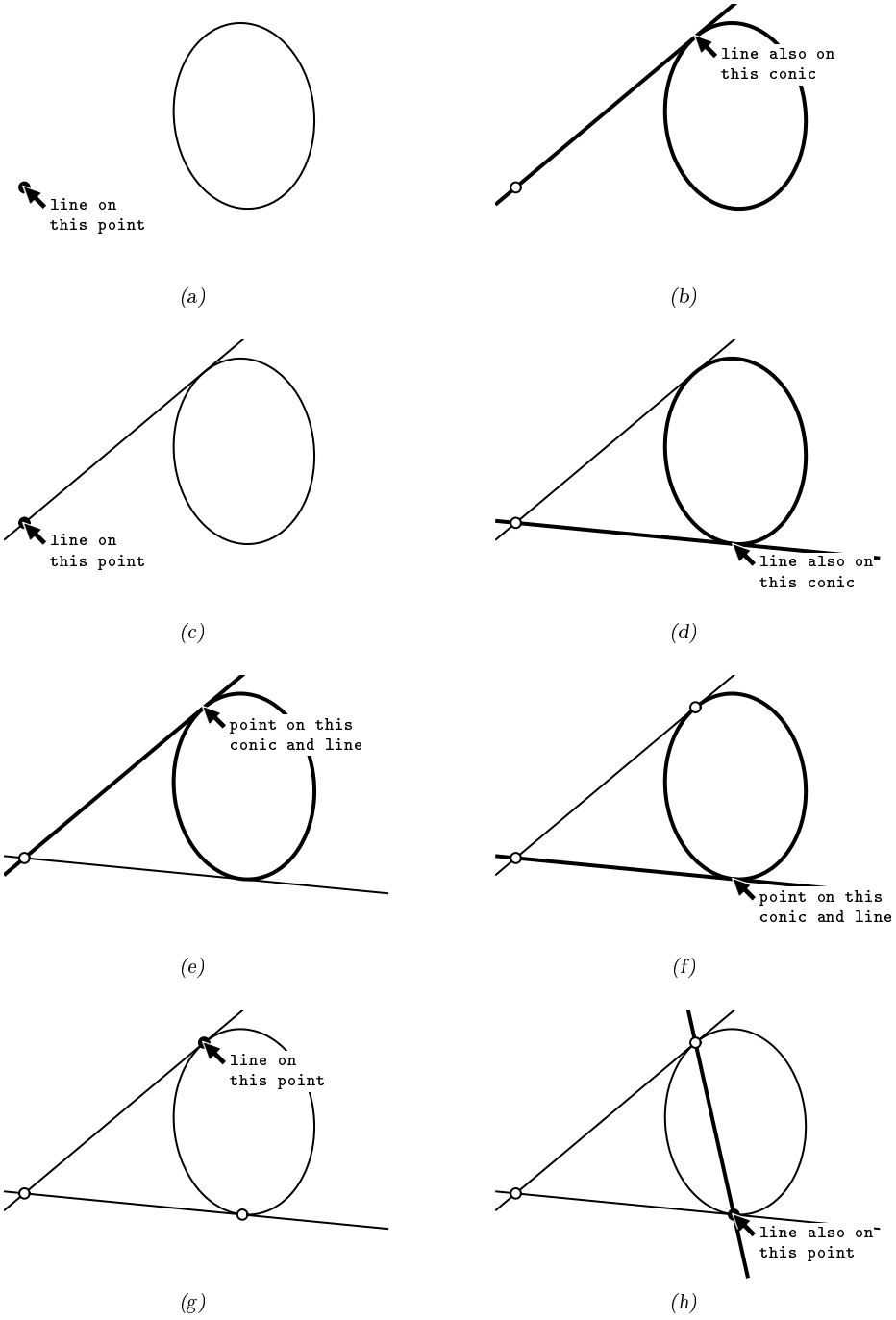


Figure 5.52. Constructing the polar of a point in pdb.

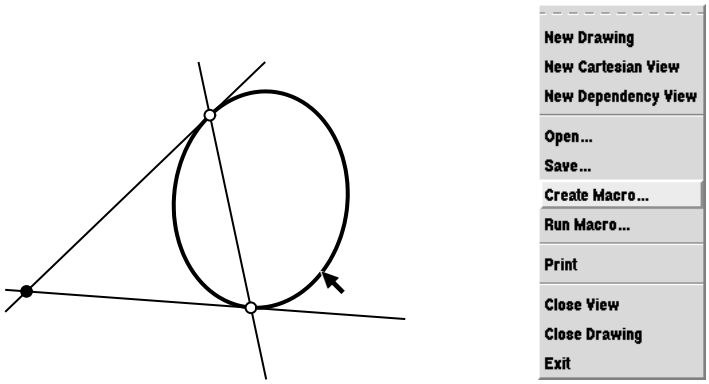


Figure 5.53. Creating a polar line macro.

`Macro` (Figures 5.53a-b). The `Create Macro` function generates Tcl code which will recreate all objects that depend directly or indirectly on the input objects. Other objects that are necessary for the construction will also be recreated by the macro[6]. The macro is stored in a plain text file, the name of which is specified by the user. The user may also choose a macro name, such as `Polar`. The macro will immediately be added to the `Macro` menu under the specified name.

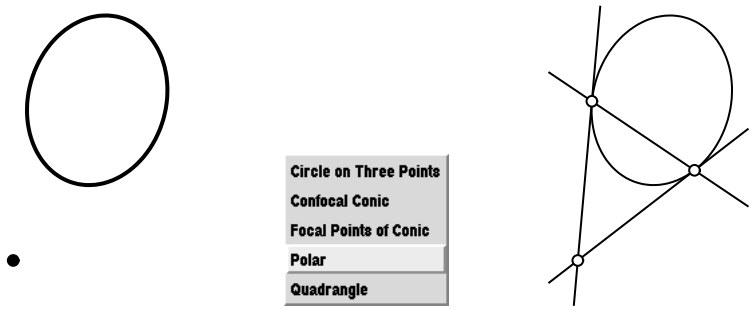
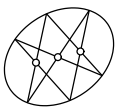


Figure 5.54. Applying the polar line macro.

Figure 5.54 shows how the polar line macro is used. A conic and a point are selected and `Macro`→`Polar` is invoked. Note that the macro has made the tangent lines and the tangent points visible in Figure 5.54c. This is because they were visible in Figure 5.53 when the macro was saved. If we had made these objects

[6] To be more specific, let S be the set of objects which depend on the macro arguments. As long as there is an object $x \in S$ which depends on an object $y \notin S$, and there is a path from y to x in the dependency graph that does not pass through one of the macro arguments, y will be added to S. The resulting set S will be recreated by the macro.



invisible before invoking `File→Create Macro`, they would have been invisible in Figure 5.54c too.

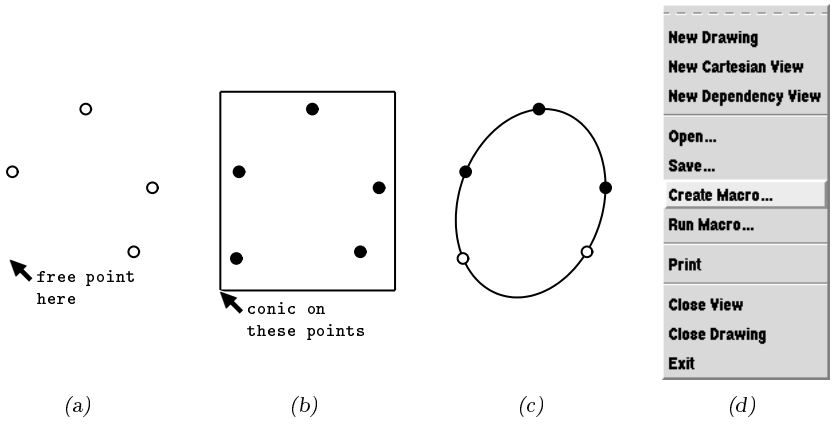


Figure 5.55. Creating a circle macro.

```
pdb_get_selection $drawing_id Point 3 points
lappend points [Point::create $drawing \
    [FreePoint::create {0.66 0.75 0.0058}]]
lappend points [Point::create $drawing \
    [FreePoint::create {0.15 0.99 0.0073}]]
set conic [Conic::create $drawing \
    [eval ConicOnFivePoints::create $points]]
View::create_image $view $conic
```

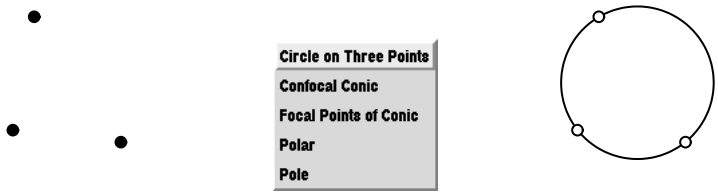
Figure 5.56. A macro that constructs a conic on five points.

Sometimes it is necessary to edit a saved macro. For example, suppose we want to define a macro which creates the Euclidean circle through three given points. As explained in Section 4.12.3, all circles in Euclidean geometry pass through the two circular points I and J whose coordinates are $(i, 1, 0)^T$ and $(-i, 1, 0)^T$, respectively. Therefore, to define the circle, we can use the built-in conic tool to create a conic on five points, then replace two of the points with I and J. Figure 5.55 shows how to proceed.

First, we create five arbitrary points, using the point tool (Figure 5.55a). Then a conic on these points is defined by the conic tool (Figure 5.55b). Since the macro will eventually be applied to only *three* points, we select three of the points and invoke `File→Create Macro` (Figures 5.55c-d). The three selected points in Figure 5.55c are the macro arguments and are assumed to be available when the macro is invoked. The two remaining points and the conic will be recreated by

```
pdb_get_selection $drawing_id Point 3 points
lappend points [Point::create $drawing \
    [FreePoint::create {{0 1} 1 0}]]
lappend points [Point::create $drawing \
    [FreePoint::create {{0 -1} 1 0}]]]
set conic [Conic::create $drawing \
    [eval ConicOnFivePoints::create $points]]
View::create_image $view $conic
```

Figure 5.57. The modified macro which constructs a circle on three points.



(a) *(b)* *(c)*

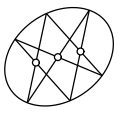
Figure 5.58. Applying the circle macro.

the macro. Figure 5.56 shows the resulting Tcl code[7]. The arguments of the two `FreePoint::create` calls in Figure 5.56 are then replaced by the coordinates of I and J, as shown in Figure 5.57. (In Tcl, a complex number $1 + 2i$ is written `{1 2}`.) Now the macro is ready to be used. In Figure 5.58a, three points have been selected, and in Figure 5.58b the macro is invoked. The result is shown in Figure 5.58c.

For convenience, a few elementary geometric constructions in Euclidean geometry are supported by a pdb macro package:

- The midpoint of a line segment
- The perpendicular bisector of a line segment
- Angle bisection
- Circle about a given center passing through a given point

[7] The code has been shortened to save space.



5.4 System design

5.4.1 Design principles and the use of object-oriented techniques

pdb was designed to be an *open-ended* system, a system which allows new types of shapes and constraints to be added later on. Also, *portability* has been a primary concern. The system currently runs under UNIX and XWindows, but a Windows95 version is planned. High performance was a very important design goal. On of the main points of writing specialized dynamic geometry software is that we can achieve a response time that cannot be match by general-purpose systems (Section 1.5).

To obtain a satisfactory design, we decided to apply object-oriented principles and to concentrate on the design of the *interfaces*. We have tried to identify and isolate the parts of pdb that are most likely to be changed or extended in the future. In particular, all platform-dependent parts, such as the underlying windowing system, have been encapsulated. On the other hand, we have been very careful not to create inefficiencies by introducing unnecessary interfaces or software layers.

The structure of the system and the responsibility of different classes will be discussed in the following sections. We will use UML diagrams [Booch99] to illustrate class relationships and object interactions, and we will point out the design patterns [Gamma95] that have been employed. The choice of programming language will be discussed in Section 5.4.3.

5.4.2 Main packages

pdb consists of three main packages: the *kernel*, the *user interface* (UI), and the *windowing system interface* (WSI), see Figure 5.59.

- The kernel handles all geometric computations. It maintains the dependency graph and makes sure that the constraints on the geometrical objects are satisfied at all times. It communicates with the UI package in order to keep the views updated and consistent. However, the kernel is largely independent of the UI package and could therefore be used in any application program which needs to perform geometrical computations.

- The UI package is responsible for maintaining the views, for responding to user actions, and for file management. The UI package creates the windows, buttons and menus comprising the user interface and provides the necessary event handlers. Whenever a geometric object on the drawing board, such as a line, is dragged or manipulated by the user, the UI package asks the kernel to compute new positions of all objects. In order to render a drawing, the UI package relies on the graphics support provided by the WSI package.

- The WSI package encapsulates the underlying windowing system, for example XWindows [Quercia93] under UNIX. A part of the package consists of

a windowing toolkit, Tk [Ousterhout94], which implements common graphical components such as frames, buttons, menus, scrollbars etc. The WSI package also provides a thin, application-specific interface to the windowing system which allows the UI package to draw almost directly into a physical window, without using the rather resource-consuming Tk toolkit. This interface provides exactly the set of graphical primitives necessary for the UI package to render the contents of the drawing board.

The WSI package shields the UI package from the windowing system. If pdb is ported to a new windowing system, only the WSI package will be affected. Since the Tk toolkit already runs on all major platforms, only a few application-specific classes in the WSI package need to be reimplemented.

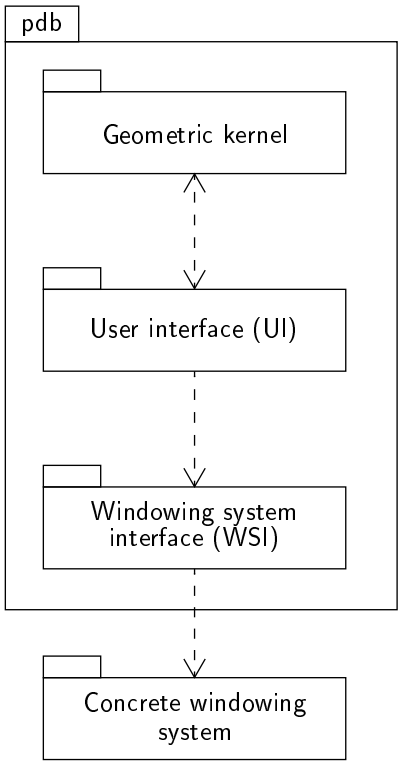
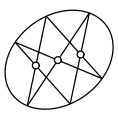


Figure 5.59. pdb consists of three main packages.

5.4.3 Choosing an implementation language

Since a short response time was a primary design goal, we needed an implementation language for the pdb kernel with strong type checking, global optimization, and in-line expansions of functions. Since the system is object-oriented, C++



[Stroustroup97] was a natural choice. The language also allows us to control the memory management in detail and to make sure that time-consuming garbage collection does not take place during user interaction.

Other implementation languages were considered, in particular Java [Arnold98]. Actually, a number of dynamic geometry demonstrations have been implemented in Java and are now available on the net [Geometry-Center98]. Cinderella Café (Section 2.3) has been written completely in Java, possibly because an earlier version was written in Objective-C, a language which in many respects is similar to Java. The Java language has many advantages; the run-time support is excellent, there are standardized toolkits for building graphical user interfaces, and Java programs can be run on most platforms with few compatibility problems. However, judging from existing Java programs, there seems to be a significant speed penalty, which cannot easily be overcome by byte-code conversions or just-in-time compilation. A slow response can be observed in all dynamic geometry systems implemented in Java, including Cinderella Café. Interestingly, the Smalltalk language [Goldberg83], from which Java has borrowed many ideas, seems to suffer from the same slowness, although its virtual machine and low-level libraries have been improved and optimized over the last twenty years. We believe that the users of pdb will be more interested in computational speed than in portability, and therefore, we decided not to use Java.

Sooner or later, users will need to make geometrical constructions that are not directly supported by the pdb kernel. In that situation, one option is of course to extend and recompile the kernel. However, that would not be practical for several reasons:

- The source code of pdb might not be available.
- The compilation time would be significant.
- Extending the kernel would involve advanced programming in C++, a language well-known for its steep learning curve.

Therefore, a special language for writing *macros* is available in pdb (cf Section 5.3.6). The macro language offers access to every function and data structure in the pdb kernel and in the UI package, but the language is interpreted and easier to use than raw C++.

The importance of having a macro language in a dynamic geometry system has frequently been pointed out, and most other geometry systems have some sort of macro facility. Instead of inventing yet another macro language, we decided to use a good, general-purpose scripting language and extend it with new commands for accessing the pdb C++ core. Obvious candidates were Perl [Wall96] and Tcl [Ousterhout94]. Since Tcl comes with the very nice, portable graphical toolkit Tk, we finally chose Tcl. The integration of Tcl and Tk allowed us to write most of the UI package in Tcl rather than in C++, which made the code very compact and easy to modify. Only the time critical parts, such as geometric computations and part of the event processing code, have been implemented in C++.

The main disadvantage of having two different languages in the same system is that many data structures and functions must be accessible from both languages and we must therefore define an interface between them. A Tcl interpreter can be extended with new, custom commands implemented in C or C++. Once the new commands have been installed into the interpreter, they cannot be distinguished from the built-in ones. However, before a C++ function can be called, the arguments must be converted from the Tcl data types to C++ objects, and when the function returns, the return value must be converted back to a Tcl data type. The conversion code is usually written by hand, a very tedious and error-prone task. For a library as large as pdb, hand-written code is not an alternative. Instead, we have developed a new system, TIDE [Winroth98], which automatically generates a Tcl/C++ interface from abstract description of the C++ functions. The problem of accessing object-oriented C++ libraries from a scripting language, andthe design and implementation of TIDE are discussed in Appendix A.

5.4.4 Type hierarchy

The dependency graph (Section 5.1.3) is a directed, acyclic graph, consisting of nodes and arcs. Although the nodes typically represent concrete objects such as points and lines, no assumptions about the meaning or the contents of the nodes are made in the part of the code that manages the graph. If there is an arc between two nodes n_1 and n_2, it simply means that n_2 somehow depends on n_1. Whenever the internal state of n_1 (typically the coordinate of the object which n_1 represents) is changed, all nodes which depend directly or indirectly on n_1 are informed and given a chance to update their internal data structures. To speed up this process, each node contains a precomputed partial ordering of all its dependent nodes. The node initiating the updating sequence uses this ordering to make sure that each dependent node will be updated exactly once and only after all its parent nodes have been updated. The sole purpose of the `Node` base type is to define this protocol (Figure 5.60).

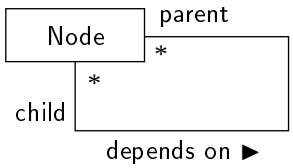
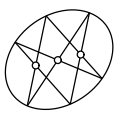


Figure 5.60. The directed, acyclic dependency graph consists of nodes.

Most of the nodes in the dependency graph will represent objects such as points, lines, conics, angles and distances. These objects are called *viewables* because they have a graphical representation (one or several *images*) on the screen. A viewable must make sure that all its images are updated whenever its internal state is changed, and it must specify what should happen if one of its images is dragged by the user. The image updating protocol and the drag-and-drop protocol are defined by the `Viewable` type, which is a subtype of `Node` (Figure 5.61).



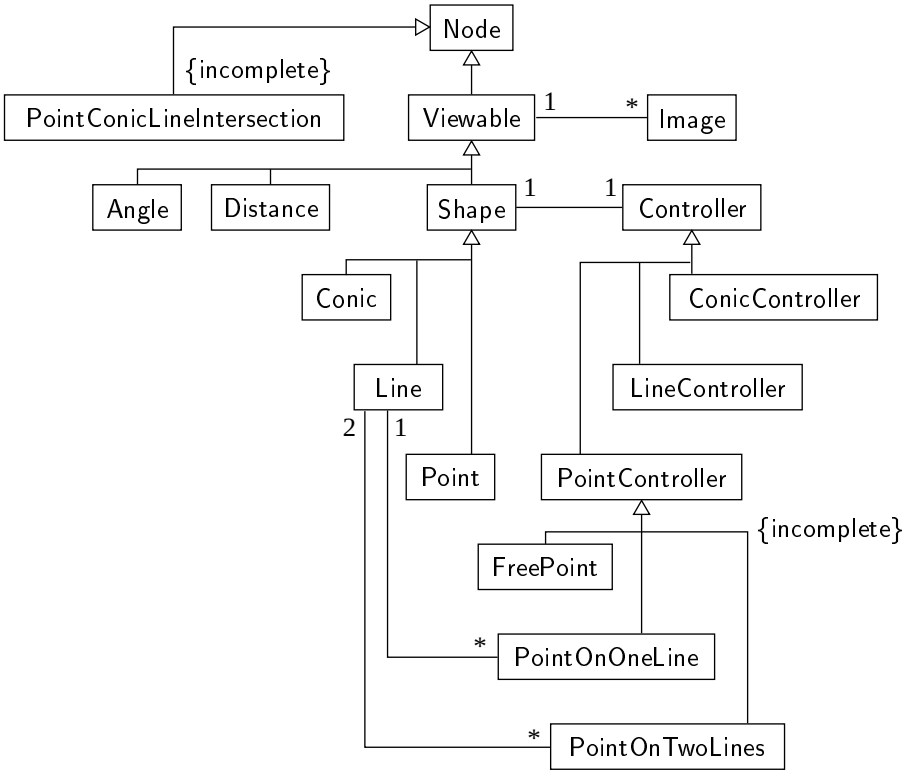


Figure 5.61. A part of the `Node` *type hierarchy.*

Certain nodes have no representation on the screen, i.e., they have no images. For example, there are nodes which just compute the roots of certain equations and cache the result. Such nodes are important for minimizing the number of times an equation has to be solved. They are represented by node types derived directly from the `Node` base type, e.g. `PointConicLineIntersection` in Figure 5.61.

As explained in Section 5.1.3, most of the nodes in the dependency graph represent a *combination* of a geometric object (point, line or conic) and a set of constraints on that object. For example, there are nodes representing free points, points attached to a line, and points attached to the intersection of two conics. If these different kinds of points were represented by different types of nodes, a node would have to be able to change its type at run-time since constraints can be added and removed dynamically by the user (Section 5.3.2). However, the type of a C++ object cannot be changed once it has been created. We have therefore been forced (for implementation reasons) to split each of these nodes into two parts. The first part is the actual node, which is one of `Point`, `Line` or `Conic`. These three node types are all derived from `Shape`, which in turn is derived from `Viewable`. The second part, which represents the constraints,

is called the *controller* and is a replaceable part of the node (see Figure 5.61). All controllers, such as `FreePoint`, `PointOnOneLine` and `PointOnTwoConics`, are derived from a common `Controller` base type. A `Shape` node will delegate a number of operations to its controller, which normally implements at least the updating and drag-and-drop protocols. Structurally, it can be seen as a Brige pattern [Gamma95]. However, the `Shape` object is configured with a certain controller by other objects, so the controller is not completely hidden inside the `Shape` class. From a behavioural viewpoint, the shape-controller cooperation can be seen as an instance of the Strategy pattern.

Nodes representing angles and distances do not need controllers since there are no constraints associated with angle and distance measurements. The node types `Angle` and `Distance` are therefore derived directly from `Viewable`.

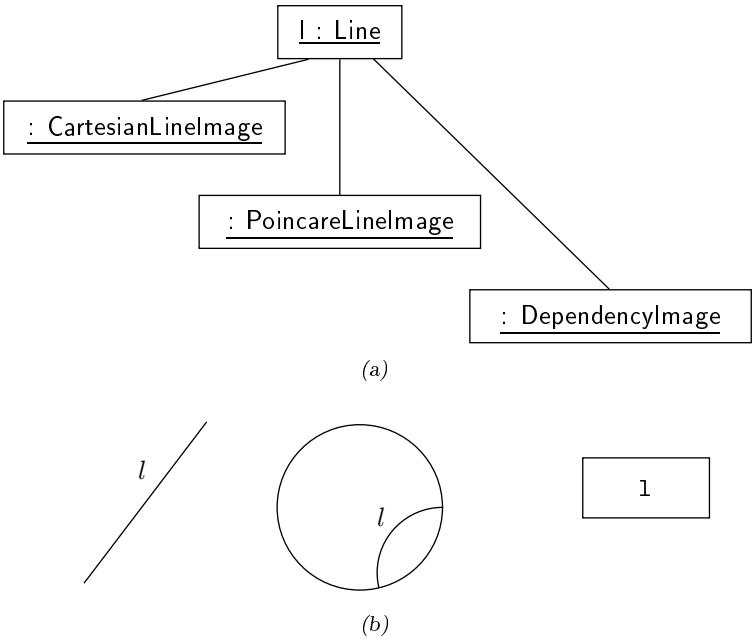
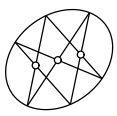


Figure 5.62. The `Line` *object is linked to three* `Image` *objects (a) which visualize the* `Line` *in three different ways (b).*

As mentioned above, there can be one or several images in different windows associated with each viewable. The image type defines how the contents of a node will appear on the screen. For example, in the object diagram shown in Figure 5.62a, a `Line` object which represents a line l is associated with three different images: a `CartesianLineImage`, a `PoincareLineImage` and a `DependencyImage`. Each type of image gives l a different appearence. The `CartesianLineImage` draws l as an ordinary, straight line, the `PoincareLineImage` draws l as a circular segment, and the `DependencyImage` displays l as a box (Figure 5.62b). An



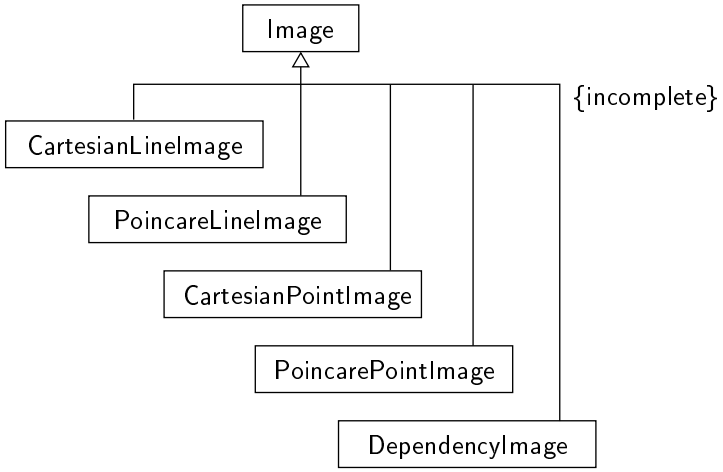


Figure 5.63. A part of the `Image` *type hierarchy.*

image and its corresponding node communicate through an interface which is defined by an abstract `Image` type. The `Image` type also has a protocol for handing user actions, such as drag-and-drop operations. Figure 5.63 shows part of the image class hierarchy.

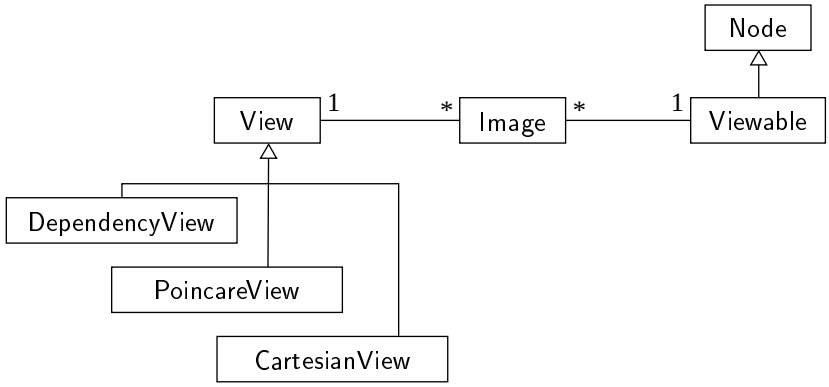
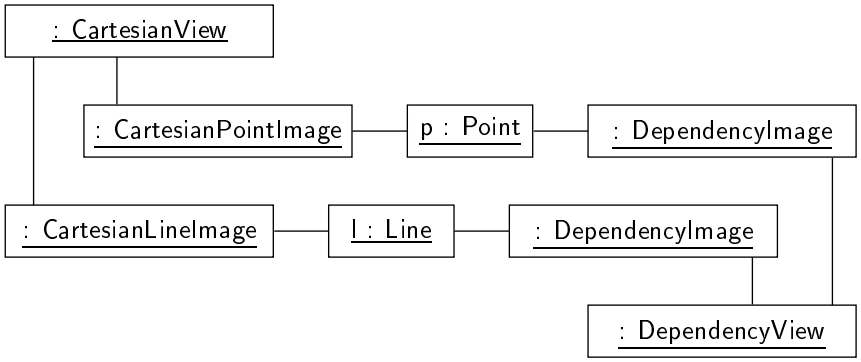
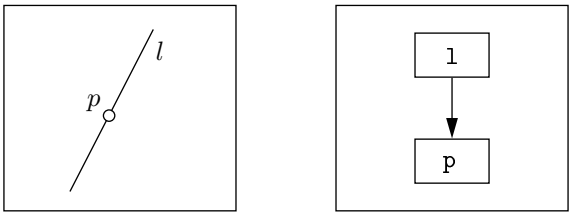


Figure 5.64. The relationship between views, viewables and images.

A *view* is basically a window in which a drawing or a part of a drawing is visible. It is represented by a `View` instance which creates and manages the physical window on the screen. Each `Image` object is associated with exactly one view and will display itself in the window belonging to that view (Figure 5.64). A viewable can have at most one image in each view, i.e. every image in a certain view belongs to a different viewable. Furthermore, all images in a view must be compatible with the view. For example, a `CartesianView` can only contain images of type `CartesianPointImage`, `CartesianLineImage` etc, while a `DependencyView` can only display `DependencyImage` instances. In Figure 5.65, a



(a) The object diagram.



(b) The two views as they appear on the screen.

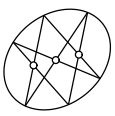
Figure 5.65. A line and a point and their images.

point p and a line l have two images each which are associated with two different views. The object diagram is shown in (a) while (b) shows the two views as they appear on the screen.

Upon request, a view will create the appropriate type of image for each type of viewable. Thus, a `View` acts as an Abstract Factory [Gamma95] for its `Image` objects. Since the type of image to be created depends on both the type of viewable and the type of view, multiple dispatch is required to invoke the appropriate factory function. Because C++ only supports single dispatch (using what is called *virtual functions*), the image creation logic has been implemented as a Vistor pattern [Gamma95].

All references to the nodes in the dependency graph are counted. A node is guaranteed to exist as long as other nodes in the graph depend on it, or as long as there are images of the node in some view (the latter applies of course only to `Viewable` nodes). This means that a node will not be removed until it is impossible to reach it or interact with it. For example, the macro for constructing a circle on three points (Section 5.3.6) creates two auxiliary nodes which represent the absolute points I and J. These points will not be visible, but they will be kept alive as long as the circle exists. When the last image of the circle is deleted by the user, the circle and the auxiliary points will be removed automatically.

A *drawing* consists of a dependency graph (the nodes and its connections),



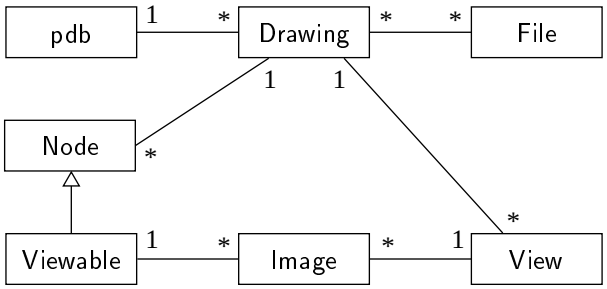


Figure 5.66. The structure of a pdb drawing.

and a set of views. A single pdb invocation can handle several drawings at the same time, but the drawings are isolated from each other. A drawing is also a unit that can be saved to file (Figure 5.66).

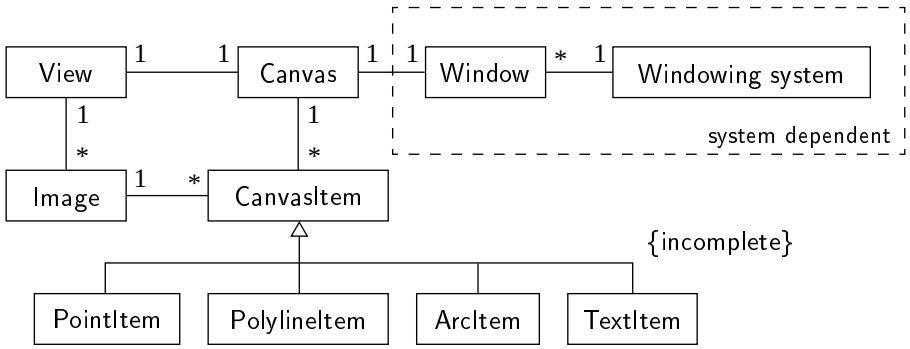


Figure 5.67. The `Canvas` *and* `CanvasItem` *types hide the underlying windowing system.*

As mentioned above, a view is associated with a physical window. The `Image` instances which belong to the view are responsible for making themselves visible in that window. In order to shield the view and the images from the details of the underlying windowing system, a physical window is represented by a `Canvas` instance, and an `Image` may create one or several `CanvasItem` instances in that canvas. A canvas item is a graphical primitive, typically a point, polyline or text item (Figure 5.67). The `Canvas` and `CanvasItem` classes provide an abstract interface to the windowing system. The implementation of the `Canvas` and `CanvasItem` interfaces are provided by hidden, system-dependent classes using a Bridge pattern [Gamma95].

A `CanvasItem` will redraw itself (e.g. when an obscured window becomes visible again) once it has been created – the `Image` object is not be responsible for doing that. Also, the properties of a `CanvasItem` (color, stacking order, etc) may be queried and changed. However, clipping operations are performed at the `Image` level since that will usually be more efficient. For example, to draw a conic, one can compute a polygon approximation and clip each line segment in the polygon to the visible part of the window. However, it is usually faster

to clip the conic itself to the window and then for each visible conic segment compute a polyline approximation which can be drawn without further clipping. Thus, instead of having a `CartesianConicImage` object create a single set of line items representing the entire conic, the `CartesianConicImage` object will create serveral smaller sets of line items which only represent the visible parts of the conic.

5.4.5 Event processing

An *event* is a record generated by the windowing system to inform an application, in this case pdb, that something potentially interesting has occurred. For example, whenever the user moves the mouse or presses a button, an event will be generated.

As explained in Section 5.4.3, most of the objects comprising the graphical user interface, in particular all menus, buttons, scrollbars etc., are part of the Tk toolkit and controlled by Tcl code. Events associated with these objects will therefore be processed by the Tcl interpreter. The Tcl event processing model is very flexible and the event bindings can be changed or extended without recompiling pdb.

Although the Tcl scripts are slower than the corresponding C++ code, they are sufficiently fast to handle events that do not occur too often, such as menu selections. However, events that occur frequently and require heavy geometrical computations are handled by C++ code. For example, if the user drags a point object on the drawing board, then for each pixel that the mouse is moved, the images of the point and all objects depending on it must be updated in all views. This will be done without any involvement of Tcl/Tk.

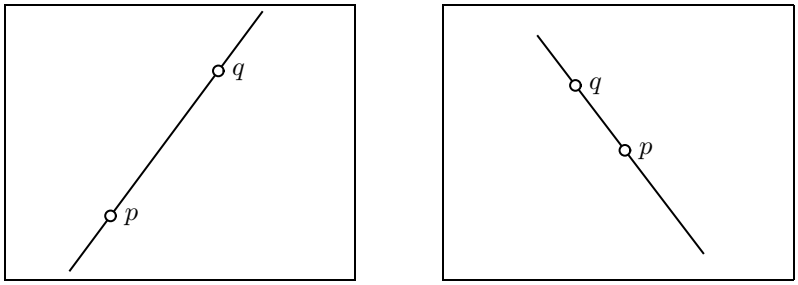
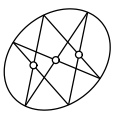


Figure 5.68. Two views of the same drawing.

Let us look at a specific event, see how it is processed and what classes are involved. Figure 5.68 shows two views of a drawing which consists of a line l attached to two points p and q. The corresponding object diagram is shown in Figure 5.69. The two points, the line, and their controllers are found at the bottom of the object diagram. The two points and the line have two images each which in turn are linked to two different views.

Assume that the user has picked the point p in view (a) in Figure 5.68 and starts dragging it. Each motion event reported by the windowing system generates



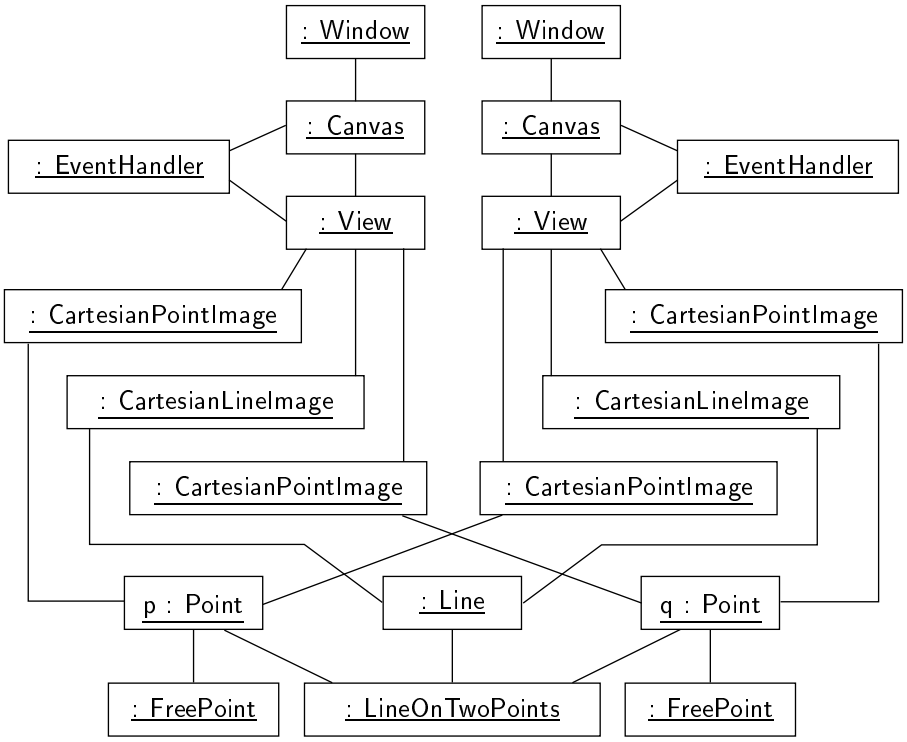


Figure 5.69. The object diagram corresponding to Figure 5.68.

a cascade of operations in the pdb object structure, as the object collaboration diagram in Figure 5.70 shows. First, the motion event is passed to the event handler for view (a). The event handler knows that p has previously been picked and interprets the motion event as a dragging operation on p. It therefore asks the `Image` object which represents p in view (a) (in this case a `CartesianPointImage`) to follow the cursor by invoking the `drag` operation on the `Image`. Depending on the constraints that have been placed on p, the `Image` object may not be able to meet this request. The `drag` call is therefore forwarded to `p`, the `Point` object representing p, which in turn passes the request on to its controller, in this case a `FreePoint` controller. Since p is a free point, the controller allows it to change its position. Because the position of the point is physically stored in the controller itself, the controller calls its own `update_coord` method to reset it. Next, `p` notifies both its images that its position has been changed. (Only one of these calls is shown in Figure 5.70.) The images, one in each view, respond by fetching the new coordinates of `p` and redraw themselves. `p` also notifies `l`, the `Line` object representing l, about the change in position, so that it can update its position too. The controller of `l` is informed and computes a new position for the line. `l` then notifies its images.

This sequence of calls will be repeated each time the cursor is moved, which

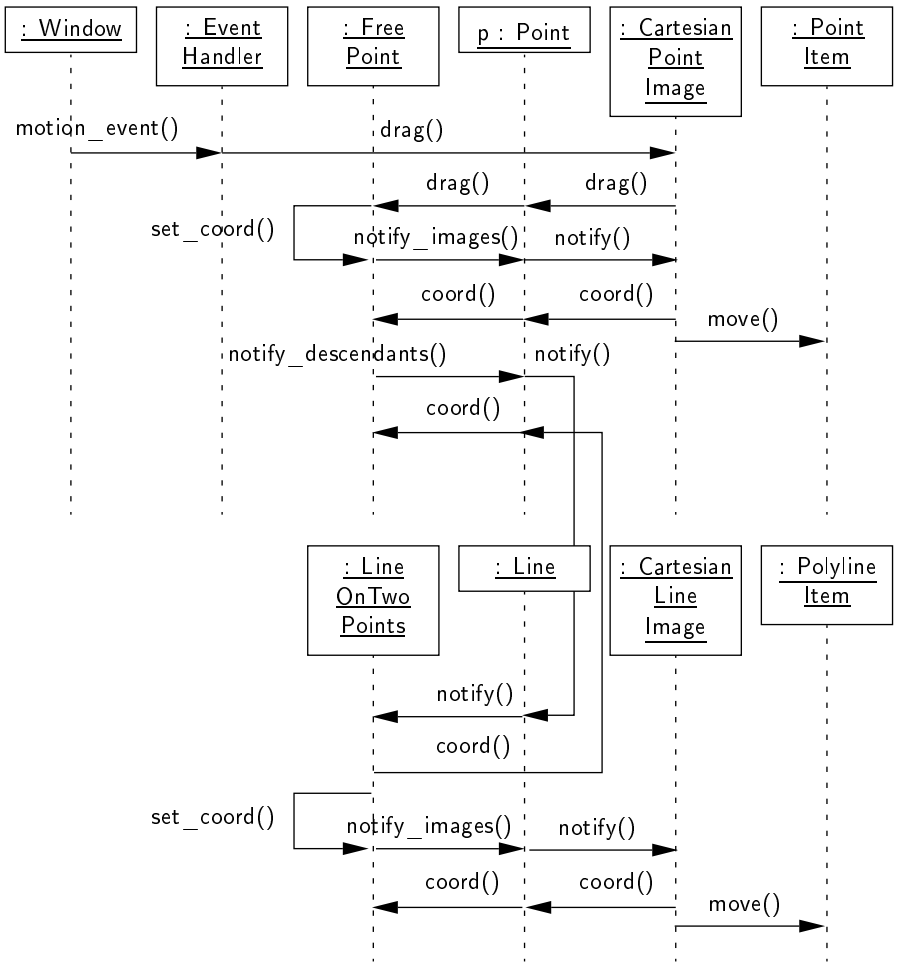
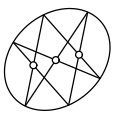


Figure 5.70. Object collaboration during motion event processing.



generates a significant amount of computation. However, the fact that all functions processing motion events have been written in C++, and that the canvases can communicate with the underlying windowing system with very little overhead cuts down the response time.

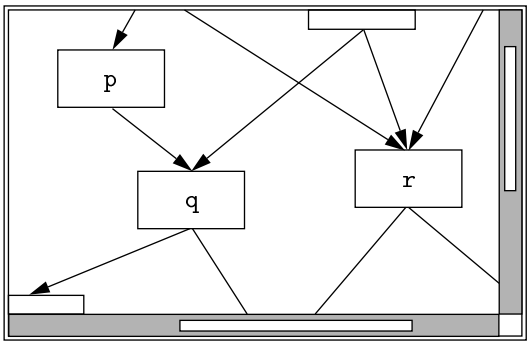


Figure 5.71. To display the scrollbars correctly, the Tcl interpreter has to know the total extent of the dependency graph.

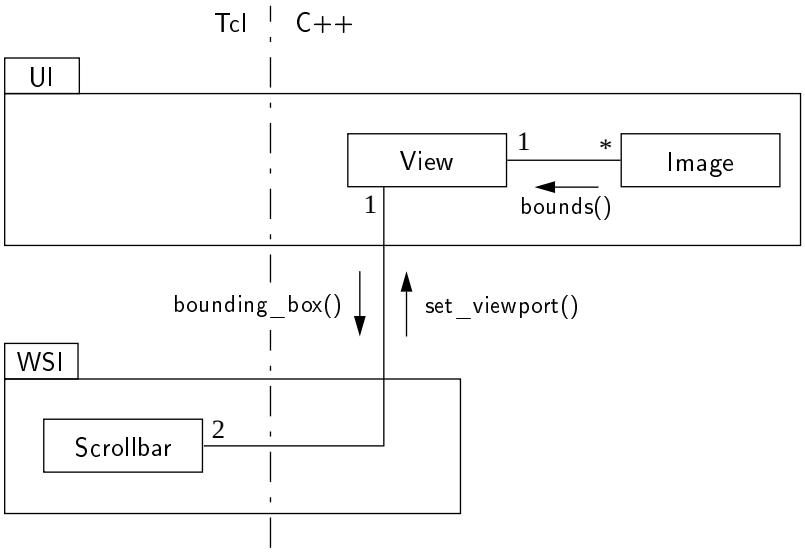


Figure 5.72. If the `View` *sends bounding box information directly to the scrollbar, the* `View` *has to know the Tcl name of the scrollbar, which makes the* `View` *dependent on Tcl.*

A complication of having event handlers partly written in Tcl and partly in C++ is that it sometimes requires bi-directional communication across the Tcl/C++ interface. For example, the window of a `DependencyView` can be scrolled using two scrollbars attached to it (Figure 5.71). Since these scrollbars are part

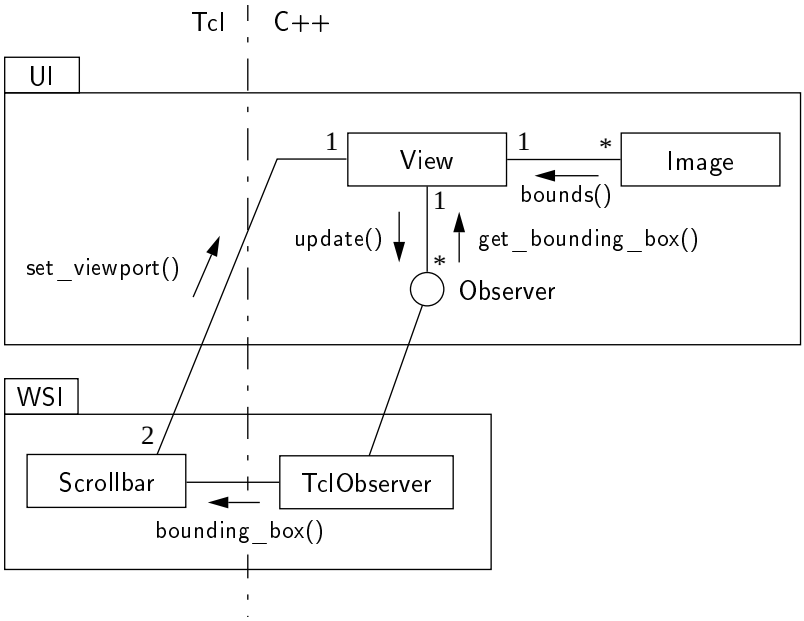
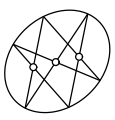


Figure 5.73. If the `View` *instead uses an abstract* `Observer` *interface, all Tcl-dependent code can be collected into a* `TclObserver` *class which implements the* `Observer` *interface. The* `TclObserver` *is responsible both for fetching the current bounding box from the* `View` *when* `update` *is called, and for passing that information on to the scrollbars.*

of the Tk toolkit, the events directed to them will be processed by Tcl code. In order to display the scrollbars and to make them operate correctly, the Tcl interpreter has to know the bounding box of the images in the view, and that box may change each time an object in the view is dragged. Since dragging operations are handled by C++ code, a C++ event handler must make the bounding box information available to the Tcl procedures which manage the scrollbars. The Tcl code can then compute how to move the viewport and pass that information to the `View` object, which is also implemented in C++ (Figure 5.72). Fortunately, the TIDE interface (Appendix A) which ties the C++ and Tcl parts together can handle such bi-directional communication. However, the problem is that in order to make it reasonably easy to replace Tcl/Tk by some other windowing toolkit in the furture, the C++ code should depend as little as possible on the Tcl code. In particular, when a C++ object passes information to the Tcl interpreter, the object should make make no explicit references to the Tcl interpreter or to the names defined in that interpreter. To accomplish that, an Observer interface [Gamma95] should be defined on the C++ side for each type of information that must be passed to the Tcl interpreter. Specialized observers, which know about Tcl names and know how to execute Tcl commands, can then be installed by the Tcl interpreter on the C++ side (Figure 5.73). The `View` will then simply send



`update` to its observers, and it will not have to know anything about the Tcl interpreter.

5.4.6 The tools

Most of the user interaction in pdb is controlled by the tools introduced in Section 5.3.2. When a tool is active, it continuously analyses what is under the cursor and suggests a suitable action, such as creating a new object or adding a constraint. The user confirms the suggested action by clicking the mouse button. The tools are the most complicated objects in pdb since they must understand what a set of constraints that has been placed on an object means geometrically.

Using Gamma's terminology, a tool is a Strategy for its view which determines how the view will react to user input. pdb comes with a set of predefined tools for creating and modifying points, lines, conics, angles and distances. However, additional tools can be easily installed. In this section we will briefly describe how a tool interacts with other objects in the system.

Each tool defines a `filter` function which, given a set of viewables (typically the viewables currently under the cursor), returns the set of objects it can act upon. For example, given a point and a line, the line tool would normally return the point, since it can create a new line on that point. However, depending on its internal state, a tool might choose to return a different subset of the input objects. For example, if the `Control` key is held down, the standard tools will attempt to modify existing objects rather than creating new ones. Given a point and a line, the line tool would then return the line if there is a constraint on the line that can be removed. For each `filter` call, the tool also returns a string describing the suggested action, e.g. `"line on this point"` or `"detach line from this point"`.

A tool is also required to support drag-and-drop actions, which are implemented by three functions: `pick`, `drag` and `drop`. `pick` will usually be invoked when the mouse button is pressed and `drop` when it is released. In between those calls, `drag` will be called each time the cursor is moved. Typically, a new object is created or constraints are removed by `pick`, the coordinates of the new or modified object are updated by `drag`, and finally, new constraints may be added by `drop`. Figures 5.39a-d, page 135, shows how the point tool suggests a pick, creates a new point, drags the point into a new position and finally adds a second constraint to it.

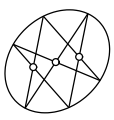
Drag-and-drop operations work well as long as only one or two objects are involved. However, some operations need several operands. For example, the conic tool must be given five point objects to create a conic through five points. In such cases, the tools interact with the current selection. If the user picks an object whose type is consistent with one of the operations that the currently active tool can perform, the object will be added to the current selection. When enough operands have been selected, the tool will operate on them. For example, the conic tool will create a conic as soon as five points or lines have been selected.

5.4.7 File management

When a drawing is saved to file, pdb will actually compile a Tcl program which, when run, will recreate the drawing with all its objects and views. The main advantage of the approach is that the pdb files are readable by humans, and that they can be modified using a standard text editor. For example, if we would like to make a drawing where some objects have to be given specific coordinates, we could first create an approximate sketch, save it and then edit the coordinates in the resulting file. A text based file format also makes it easy for other programs to generate drawings that can be loaded into pdb.

For a large CAD drawing with thousands of elements, this approach would not be appropriate, since the resulting Tcl program would be extremely long and the load time would be unacceptable. For such massive data structures a binary file format would be more suitable. In order to keep the system open-ended and extensible, one would probably use some kind of *persistent object* scheme, which allows the file format to evolve over time while maintaining backward compatibility. One such system, which gives minimal file size overhead has been presented in [Winroth94].

However, for small- to medium-sized drawings, the advantages of the scripting approach are compelling. Furthermore, since the part of pdb which saves a drawing is also written in Tcl, it is easy to extend the information stored on file if necessary. In fact, the file format can be extended by the users, without recompiling pdb.



Chapter 6

Dynamic geometry at work

The previous chapter contained several figures illustrating various aspects of the user interface. However, since the discussion focused on user interaction and the system response, most of the examples were small and rather fragmented. One of the few complete constructions was the polar line shown in Figure 5.52, page 151. A larger set of drawings were given in Chapter 4, most of which had been generated by pdb. However, we gave no indication of how they had been constructed; only the final output was printed.

The purpose of this chapter is to give the reader some idea of how the properties of a geometric construction can be investigated and how a geometrical theorem can be illustrated using the pdb dynamic geometry system. In contrast to previous chapters, we will here provide complete examples of geometrical constructions in pdb. We will not just show the final result but describe how the drawings are actually constructed. In the first examples, we will lead the user step-by-step through the construction process. However, to save space and to avoid boring the reader, subsequent examples will be more sketchy.

Of course, there is a vast number of geometrical theorems that could be illustrated. We only have room for a few examples. The ones we have chosen come from projective, affine, Euclidean and hyperbolic geometry. They show how the different geometries are related and at the same time they demonstrate many of pdb's capabilities.

6.1 Examples from projective geometry

6.1.1 Poles and polars

In Section 4.8.5 we mentioned that a conic could be considered as a polarity, and in Section 5.3.6 we showed how the polar of a point with respect to a conic could be constructed geometrically in pdb (Figure 5.52, page 151). The construction was saved as a macro, `Polar`, in Figure 5.53. There is also a standard macro in pdb called `Pole` for the dual construction, the pole of a line. Let us verify

experimentally the symmetry property discussed in Section 4.5 using these two macros.

Given a conic C and a point p, create the polar l by selecting p and C and applying the `Polar` macro (Figures 6.1a-c). Activate the point tool and create a point q attached to l (Figures 6.1d-e). Create the polar m of q by reactivating the selector tool, selecting q and C and applying the `Polar` macro again (Figures 6.1f-i). We can see that m is on the original point p, and it will stay on p even if p or q is dragged (Figure 6.1j).

6.1.2 Verifying the self-polar property of the diagonal triangle

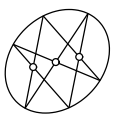
In Section 4.10 we introduced quadrangles, diagonal points and diagonal triangles. We saw that the diagonal triangle is self-polar with respect to any conic that contains the four vertices of the quadrangle. Let us check that. We will create a quadrangle and a conic on four arbitrary points. Then we will apply the `Polar` macro to each diagonal point to verify that the corresponding polars really are the sides of the diagonal triangle.

Activate the point tool and place four arbitrary points on the drawing surface (Figure 6.2a). These will be the vertices of the quadrangle. Switch to the line tool and draw the six sides of the quadrangle (Figures 6.2b-f). This quadrangle construction might come in handy later, so let us save it as a macro. We simply select the four base points and choose `File`→`Create Macro` (Figures 6.3a-b).

Using the point tool, attach a point to each intersection of opposite sides (sides that do not share a vertex of the quadrangle). These three points are the diagonal points (Figures 6.4a-b). Now, let us create the conic. Since it takes five points to define a conic, we place an additional point on the background. Then, with the conic tool active, we pick the four vertices and the free point (Figures 6.5a-c). Select the conic and one of the diagonal points and invoke the `Polar` macro. Repeat that for the reaming two diagonal points (Figures 6.5d-g). As expected, the polars form a triangle whose vertices are the diagonal points of the quadrangle. By moving the fifth, free base point of the conic we can verify that the shape of conic does not matter, as long as it contains the four vertices of the quadrangle (Figure 6.5h).

6.1.3 Creating the harmonic conjugate

Each side of the diagonal triangle of a quadrangle intersects the sides of the quadrangle in four point points, two of them being diagonal points. We saw in Section 4.10 that the cross-ratio of these four intersection points is -1, i.e., the points constitute a harmonic set. Let us first verify that. We continue to work with Figure 6.5 and define two additional intersection points on one of the sides of the diagonal triangle, as shown in Figure 6.6a. Then we select the four points and invoke the `Cross-ratio` macro (Figures 6.6b-d). As expected, the cross-ratio is -1. Note that the order in which the points are selected in Figure 6.6b is



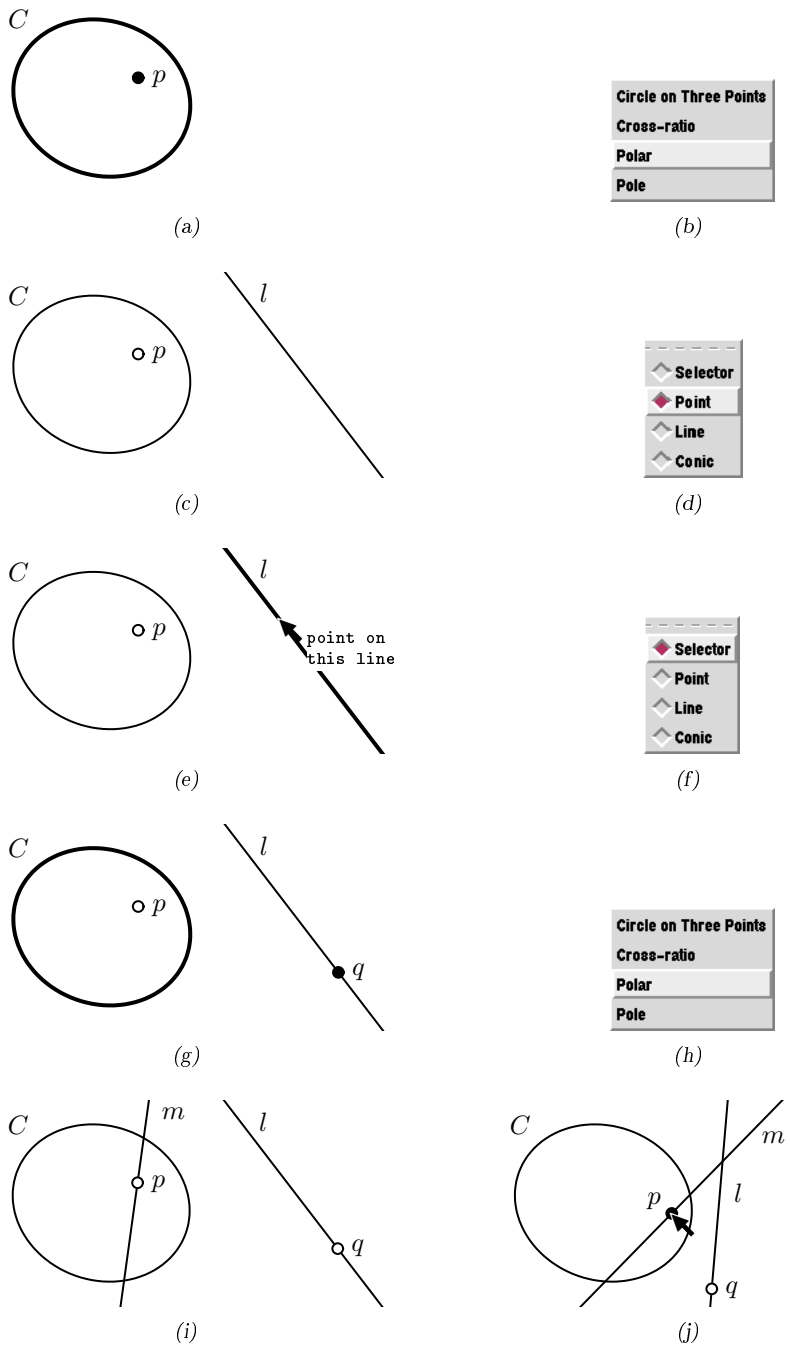


Figure 6.1. The symmetry properties of poles and polars.

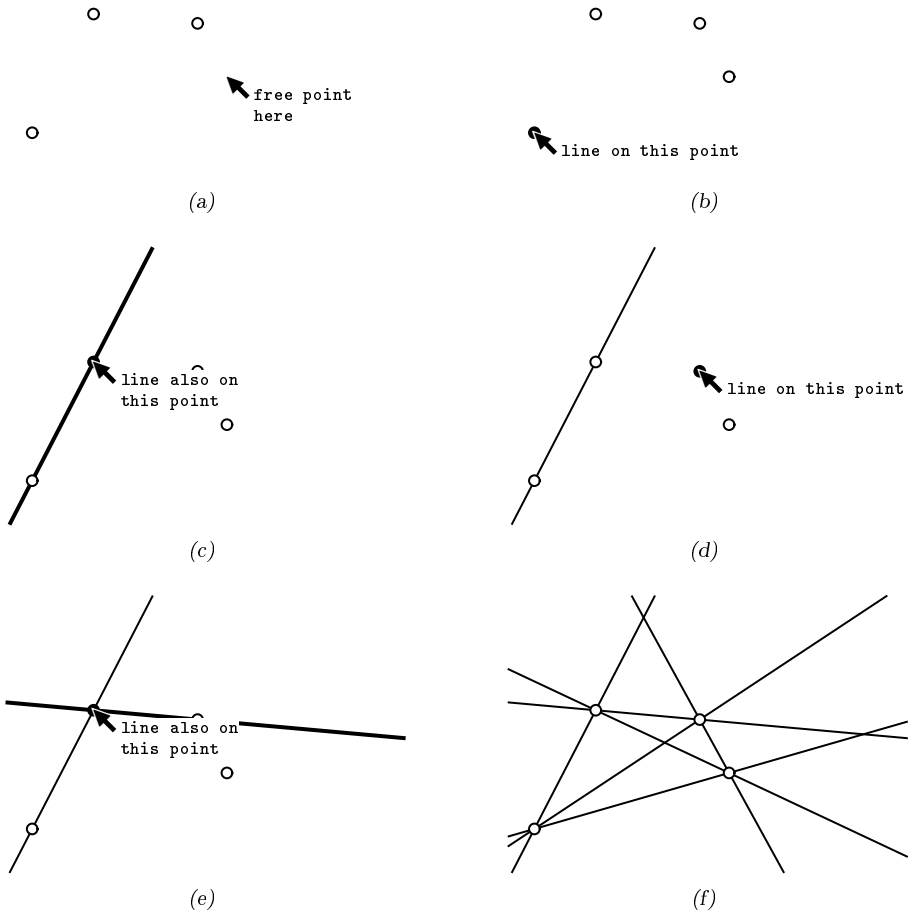
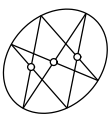
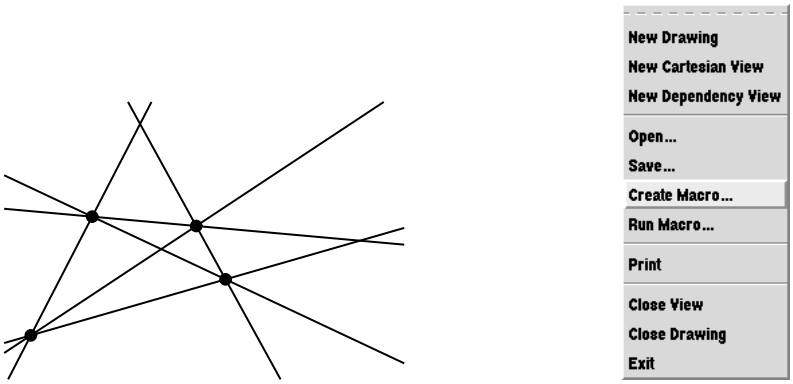


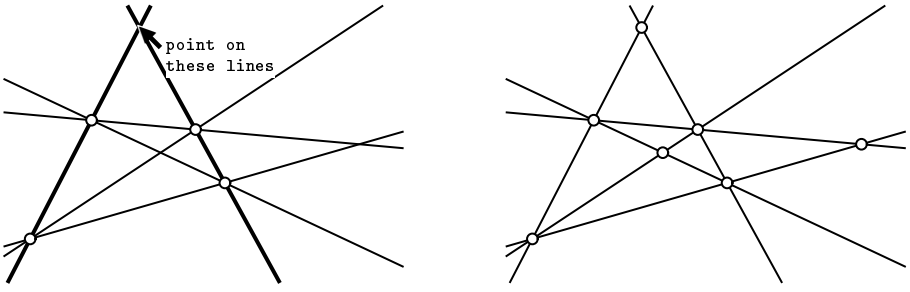
Figure 6.2. Constructing a quadrangle.





(a) (b)

Figure 6.3. Recording the quadrangle construction as a macro.



(a) (b)

Figure 6.4. Adding the diagonal points.

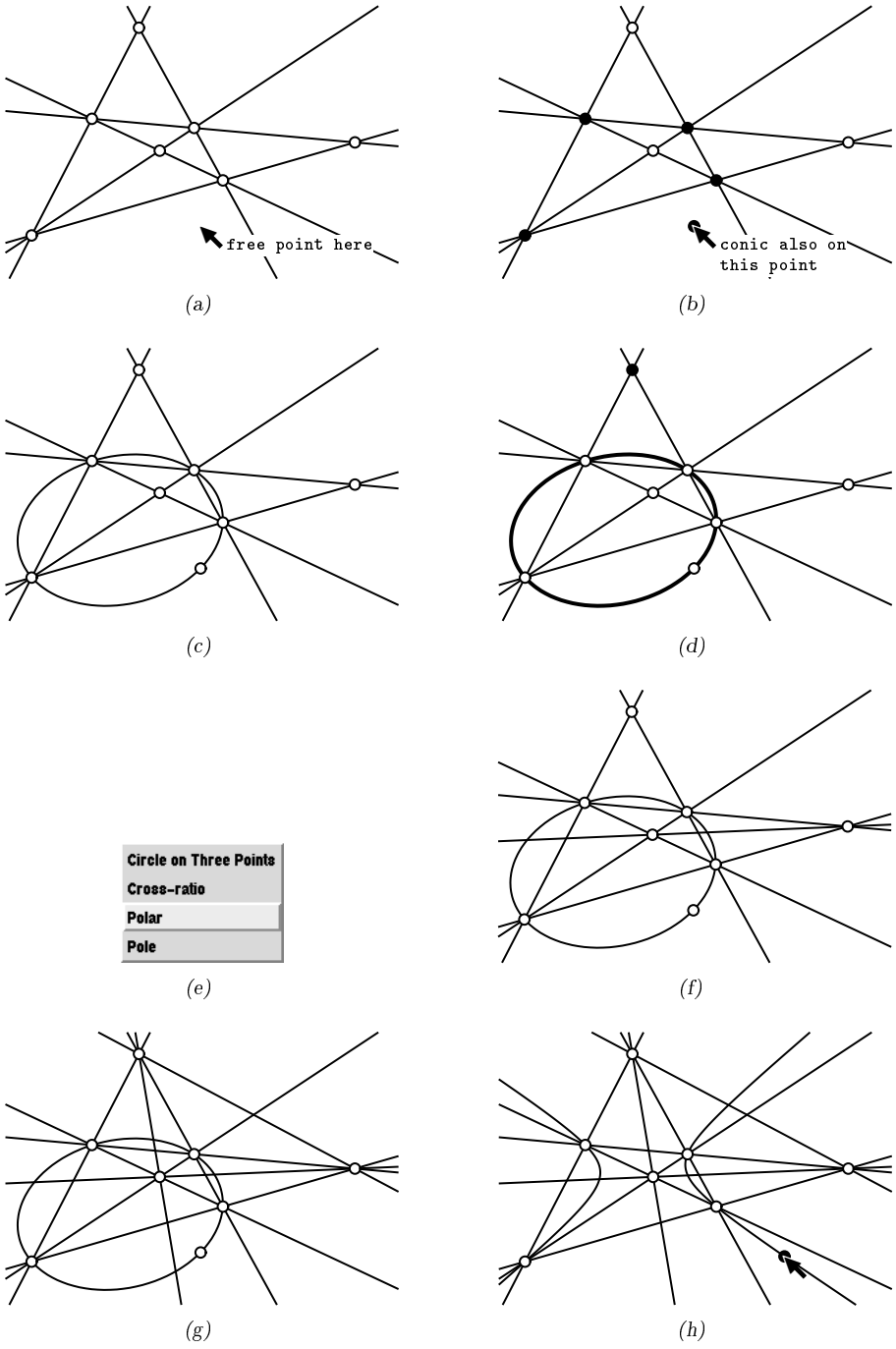
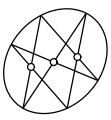


Figure 6.5. Verifying that the diagonal triangle is self-polar.



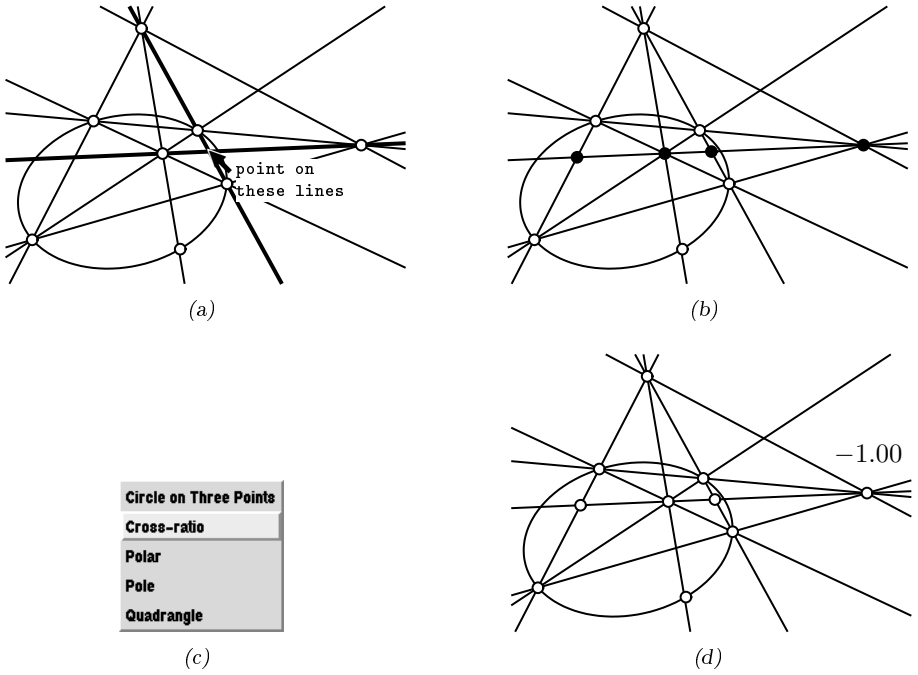


Figure 6.6. Verifying that two vertices of the diagonal triangle are harmonic conjugates with respect to the conic.

important, since it will affect the cross-ratio. If four points are selected in the order p_1, p_2, p_3, p_4, the `Cross-ratio` macro computes $(p_1\ p_2 \mid p_3\ p_4)$.

Actually, given three collinear points p_1, p_2, p_3 we can use a quadrangle to *construct* the harmonic conjugate of p_3 with respect to p_1 and p_2. That is, we can create a point p_4 so that $(p_1\ p_2 \mid p_3\ p_4) = -1$. Proceed as follows. First, create the three starting points p_1, p_2, p_3. To make sure they are collinear, we can attach them to a line l (Figure 6.7a). Define an arbitrary point q_1 not on l, and draw q_1p_1, q_1p_2, q_1p_3 (Figure 6.7b). Choose a point q_2 on q_1p_1 and draw q_2p_2 (Figure 6.7c). Let q_3 be the intersection of q_2p_2 and q_1p_3 (Figure 6.7d). Draw q_3p_1 and let q_4 be the intersection of that line and q_1p_2. The intersection of q_4q_2 and l will be the harmonic point p_4 (Figure 6.7e). We can drag any of the points p_1, p_2, p_3 and see how p_4 is moved (Figure 6.7f).

This construction can be saved as a macro and applied to any three collinear points. However, there are some pitfalls. First, p_4 was defined as the intersection of q_4q_2 and l. But l is a parent of p_1, p_2, p_3 that we created just to make sure the points are collinear. The line l will be completely irrelevant when the macro is applied to three other collinear points. We make the following modification of the construction. Select l and p_4 and invoke `Edit`→`Delete Image` (Figures 6.8a-b). Then define the line through p_1 and p_2 (which will be *derived* from p_1 and p_2) and attach a point (p_4) to the intersection of that line and q_4q_2 (Figures 6.8c-d). The

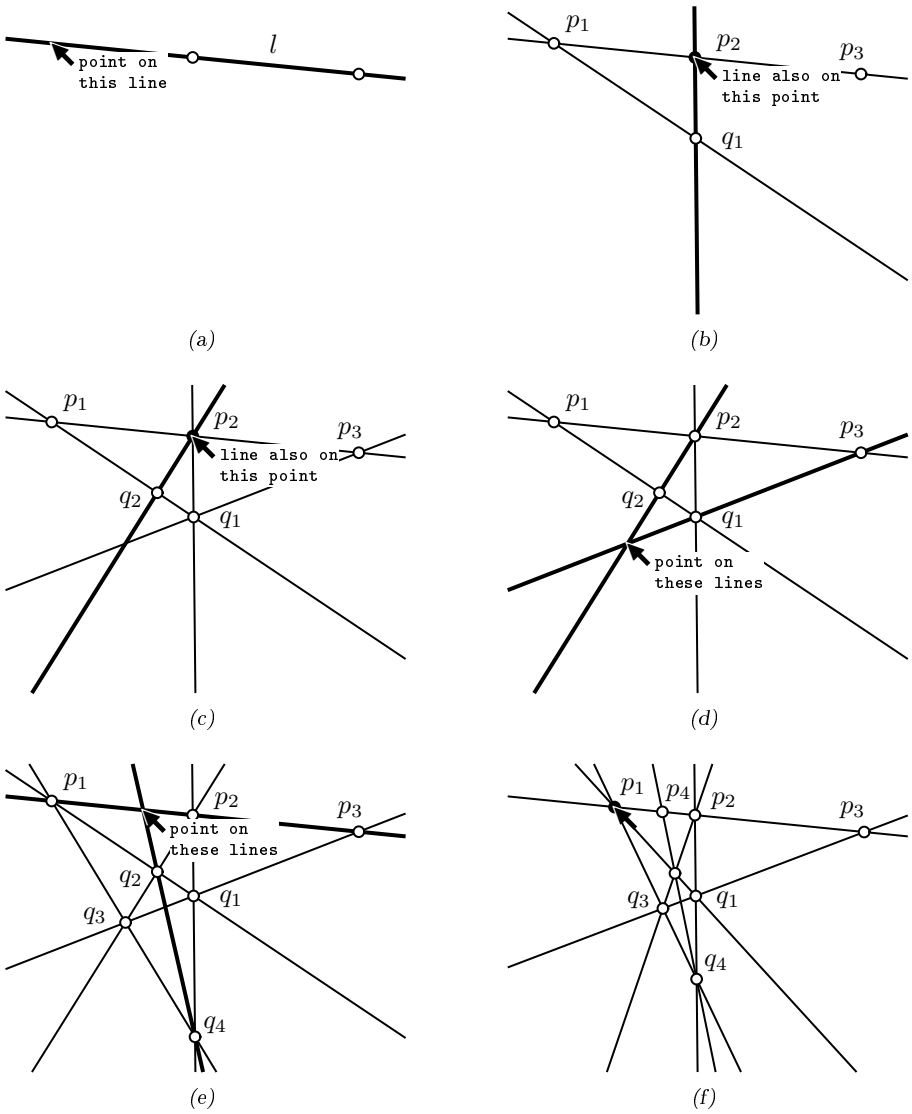
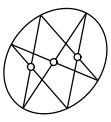


Figure 6.7. Constructing the harmonic conjugate.



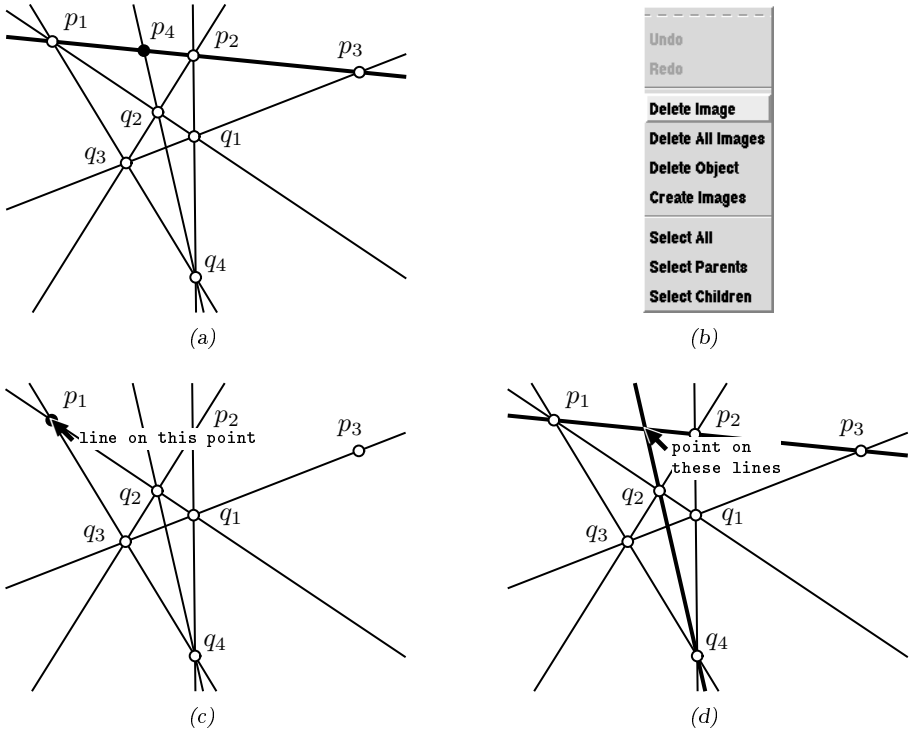


Figure 6.8. Making sure that p_4 is on a line which depends on p_1 and p_2.

second problem is that pdb macros can only be defined from constructions which contain completely free or completely constrained objects. In the construction above, q_2 was defined as a arbitrary point on q_1p_1, and that will not be accepted. The reason for this is that pdb will not know where to place q_2 on q_1p_1 when this macro is applied to a completely different set of starting points. Therefore, pdb leaves that decision to the user by requiring the position of the point to be fully specified. In this case, the placement of q_2 does not matter since it will not affect the position of p_4. We can therefore work around the problem by creating a free line intersecting q_1p_1 and attach q_2 to that intersection (Figure 6.9).

Now the macro can be defined. Since we do not want the macro to draw all auxiliary points and lines, we create a new view in which only the three starting points and the resulting harmonic conjugate are visible (Figures 6.10a-c). Then we select the three starting points and invoke `File→Create Macro` (Figures 6.10c-d). The resulting macro can now be applied to any three collinear points. We will use the macro later in this chapter.

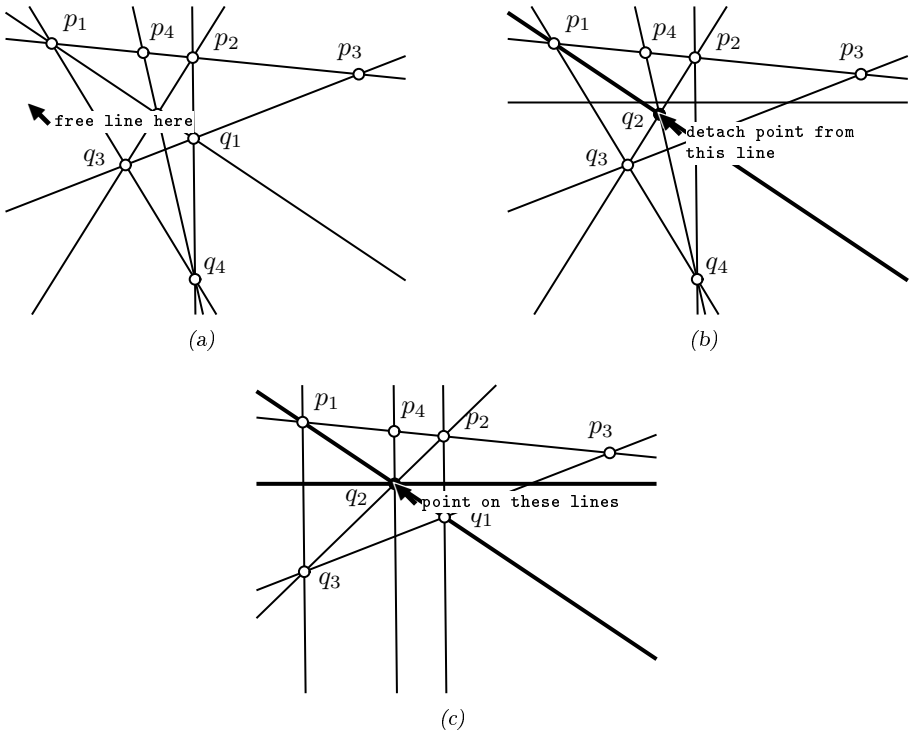


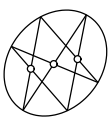
Figure 6.9. Attaching q_2 to a free, arbitrary line.

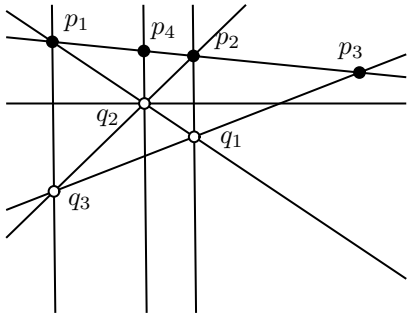
6.1.4 Trilinear polarity

Consider the triangle defined by the three vertices in Figure 6.11a. For each side s of this triangle, we will construct the harmonic conjugate of the point in which s intersects an arbitrary, fixed line l. The lines connecting these harmonic points with the opposite vertices of the triangle turn out to be concurrent.

We draw the sides of the triangle and create the three points in which the sides intersect l (Figures 6.11a-c). For each side s, we select the two vertices on s and the point in which s and l intersect. We then invoke the `Harmonic Point` macro that was defined in the previous section (Figures 6.11d-e). The lines connecting these harmonic conjugates with the opposite vertices can now be constructed (Figure 6.11f). As you can see, these lines are concurrent (Figure 6.11g). This property is true in general, which can be verified experimentally by dragging the vertices of the triangle or the line l in Figure 6.11g.

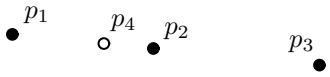
The proof of this theorem is quite simple. The original triangle p_1, p_2, p_3 can be inscribed in a quadrangle defined by p_1, p_2, p_3, p_4, as shown in Figure 6.11h. The vertices of the corresponding diagonal triangle is q_1, q_2, q_3. From Section 4.10 we know that the four points $r_1 r_3 q_1 q_2$ are harmonic. By projecting these points from q_3 onto the line $p_1 q_2$ and onto the line $p_1 q_4$, we see that both p_1, p_2, r_2, q_2





(a)

(b)



(c)

(d)

Figure 6.10. Creating the harmonic conjugate macro.

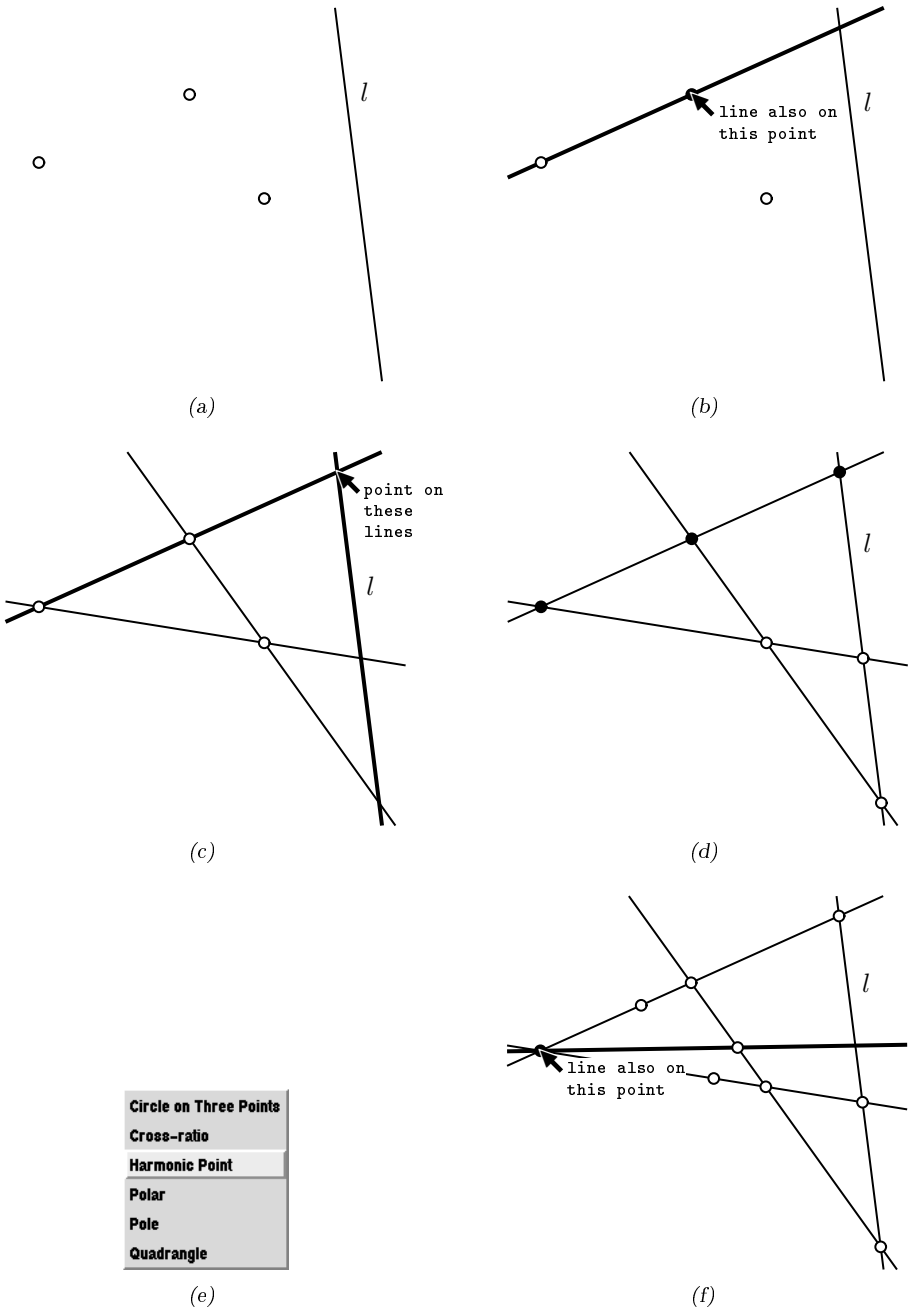
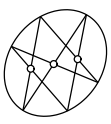


Figure 6.11. The lines connecting the harmonic points on each side with the opposite vertices are concurrent.



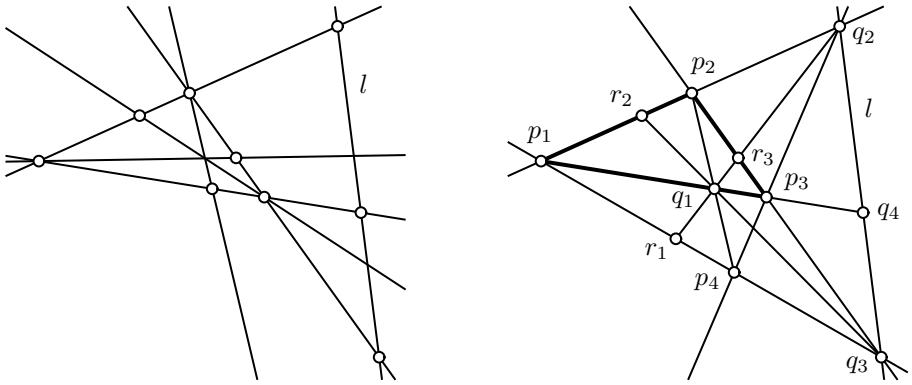


Figure 6.11. Continued.

and p_1, p_3, q_1, q_4 are harmonic. Similarly, by projecting p_1, p_3, q_1, q_4 from q_2 we verify that p_2, p_3, r_3, q_3 are harmonic. Thus, q_1, r_2, r_3 are the harmonic points we constructed in Figures 6.11d-g. Finally, by projecting p_2, p_3, r_3, q_3 onto the line p_2p_4 from p_1 and by projecting p_1, p_2, r_2, q_2 onto the same line from p_3, the theorem follows.

The point q_1 in Figure 6.11h is called the pole of the line l with respect to the *trilinear polarity* defined by the give triangle $p_1p_2p_3$. Of course, we could just as well have started with the pole q_1 and constructed the polar l.

6.1.5 Common tangents of two conics

In this section, we will investigate the common tangents of two conics. In particular, we will look at the lines connecting opposite tangent points.

The easiest way of drawing a conic is to place five points on the drawing board, activate the conic tool, and enclose the points (Figures 6.12a-b). With the line tool active, we can then create the four common tangents (Figures 6.12c-e). Next, we create the eight tangent points, four on each conic (Figure 6.12f). The lines connecting opposite tangent points are drawn in Figures 6.12g-i. Evidently, the lines are concurrent.

Why is that? Figures 6.12j-k, which shows the points in which the tangents intersects, provides a clue. Apparently, the lines connecting opposite intersection points are also concurrent with the other lines. If we compare Figure 6.12k with Figure 4.27 on page 56, we see that the four common tangents form a quadrilateral and that the point in which all lines intersect in Figure 6.12k is a vertex in the corresponding self-polar triangle. The fact that lines connecting opposite tangent points meet in this vertex follows from the properties of a polarity.

In Figure 6.12l, we have drawn the points in which the two conics intersect. Interestingly, the two lines connecting opposite intersection points are also concurrent with the rest of the lines.

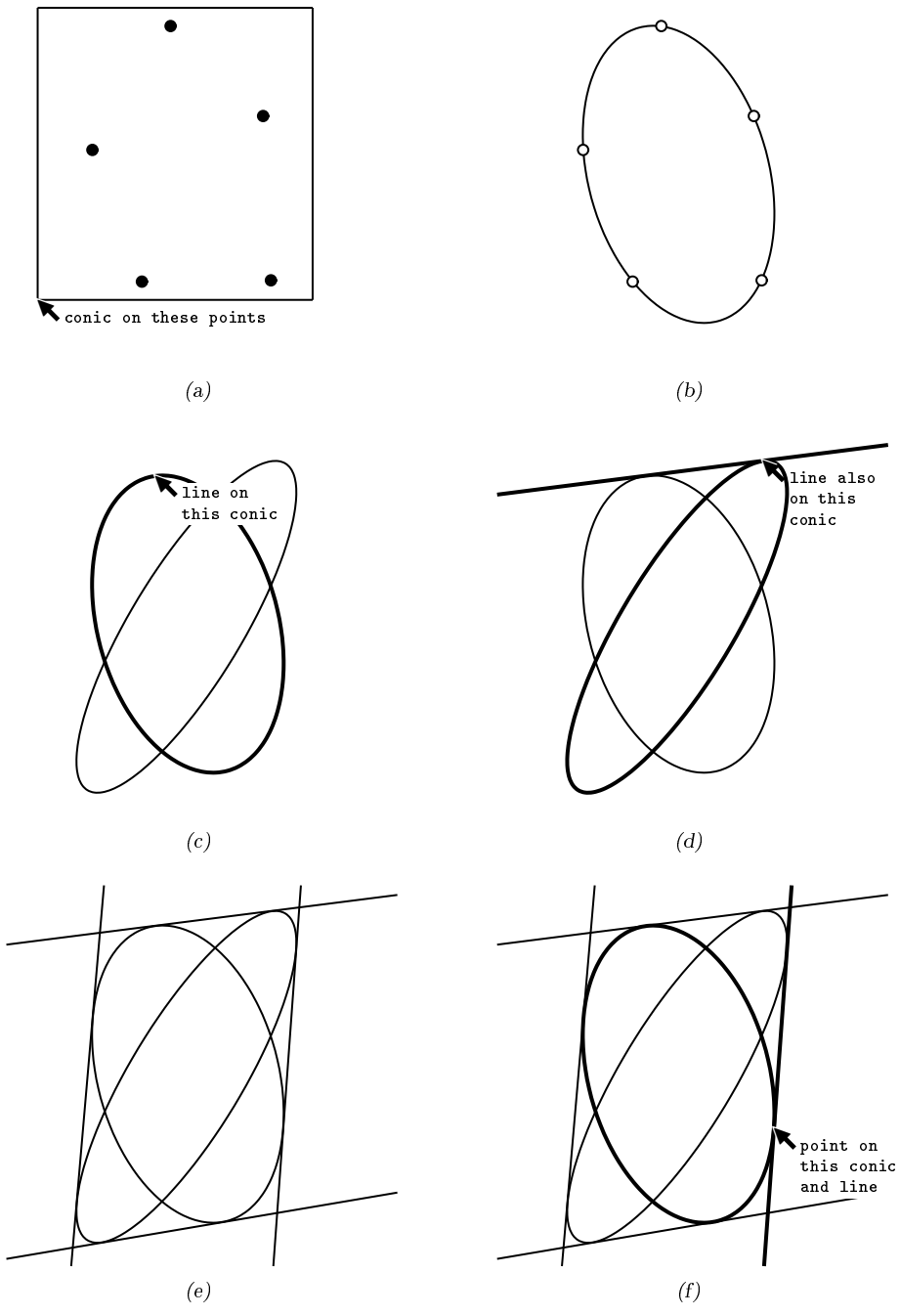
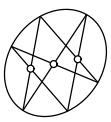
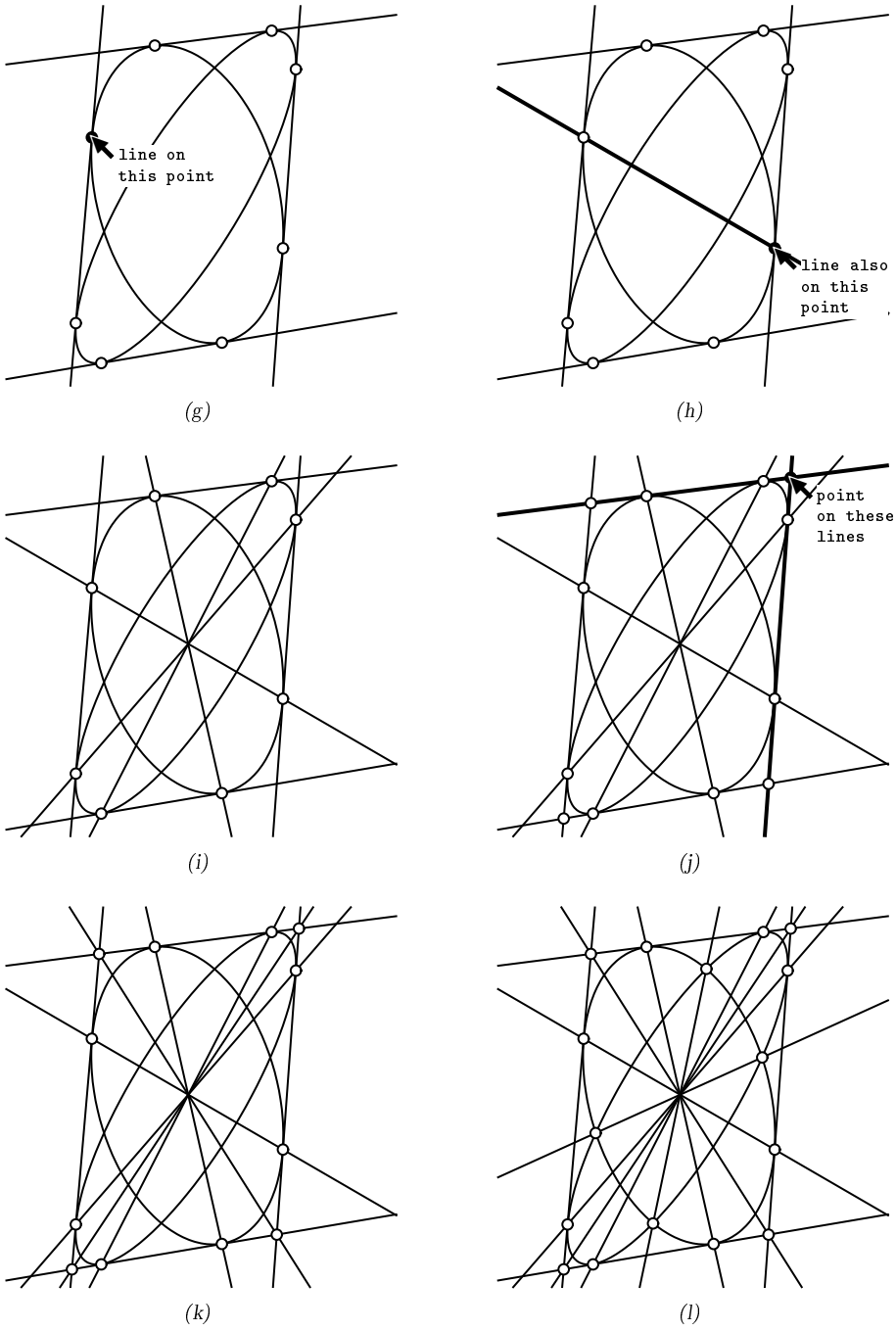


Figure 6.12. The lines connecting opposite tangent points are concurrent.





(g) (h)

(i) (j)

(k) (l)

Figure 6.12. Continued.

6.1.6 Mapping points on the projective line

In Section 4.7 we showed that any three distinct, collinear points can be mapped to three other collinear points by at most three successive perspectivities. Furthermore, if point triples are on distinct lines, two perspectivities suffice (see Figure 4.10, page 36). Let us carry out this construction in pdb.

To define the mapping, we need three pairs of corresponding points: $p_1 \mapsto p'_1$, $p_2 \mapsto p'_2$, and $p_3 \mapsto p'_3$. Given a fourth point, p, we shall construct its image, p'. We first create two arbitrary lines, place p_1, p_2, p_3, p on the first one and p'_1, p'_2, p'_3 on the second one, see Figures 6.13a-b. To simplify the sketch, we have removed the images of the two auxiliary lines in Figure 6.13c.

We now create the lines $p_1p'_1$, $p_2p'_2$, $p_3p'_3$, and $p_3p'_1$. A point q_1 is attached to the intersection of $p_1p'_1$ and $p_2p'_2$, and a point q_2 is attached to the intersection of $p_3p'_3$ and $p_2p'_2$ (Figure 6.13d). The line q_1p is created and a point r is attached to the intersection of that line and $p_3p'_1$. The final step in creating a line from q_2 to r is shown in Figure 6.13e. Finally, we define the line $p'_1p'_2$ and let the point p' be the intersection of that line and q_2r (Figure 6.13f). By dragging p to p_1, p_2 and p_3, we can verify that we have indeed constructed the desired mapping (Figure 6.13g).

To save this construction as a macro, select $p_1, p_2, p_3, p'_1, p'_2, p'_3, p$ and choose `Create Macro` from the `File` menu (Figures 6.13h-i). As always, the order in which the points are selected is significant. To prevent the macro from displaying all the auxiliary points and lines when it is invoked, we have deleted the images of all objects except the seven input points and the and resulting point p' (Figure 6.13h).

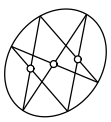
It is just as easy to define the dual construction, starting from three pairs of corresponding lines. We would then create a *point* in each step where we created a line above, and vice versa. Alternatively, we could load the macro file into a text editor and swap all strings "Point" and "Line". The resulting dual macro, which we will name `Project Line`, will be used in the next section.

6.1.7 Steiner's theorem

Steiner's theorem (Section 4.8.12) states that a conic can be defined from two pencils of lines that are related by a projectivity on P_1. The points of the conic are the intersection points of corresponding lines (Figure 4.18a, page 47). Let us verify that experimentally.

Draw an arbitrary conic and attach two points p and q to it. Create three lines on p and place a point on the intersection of each of these lines and the conic (Figure 6.14a). Connect q with each of the intersection points from (a), as shown in Figure 6.14b.

There is a unique P_1 projectivity that maps the three lines on p onto the three lines on q. To see the effect of that projectivity, create a fourth line l on p, select the seven lines and apply the `Project Line` macro that we defined in the previous section, see Figures 6.14c-d. As shown in Figure 6.14, l and its image l'



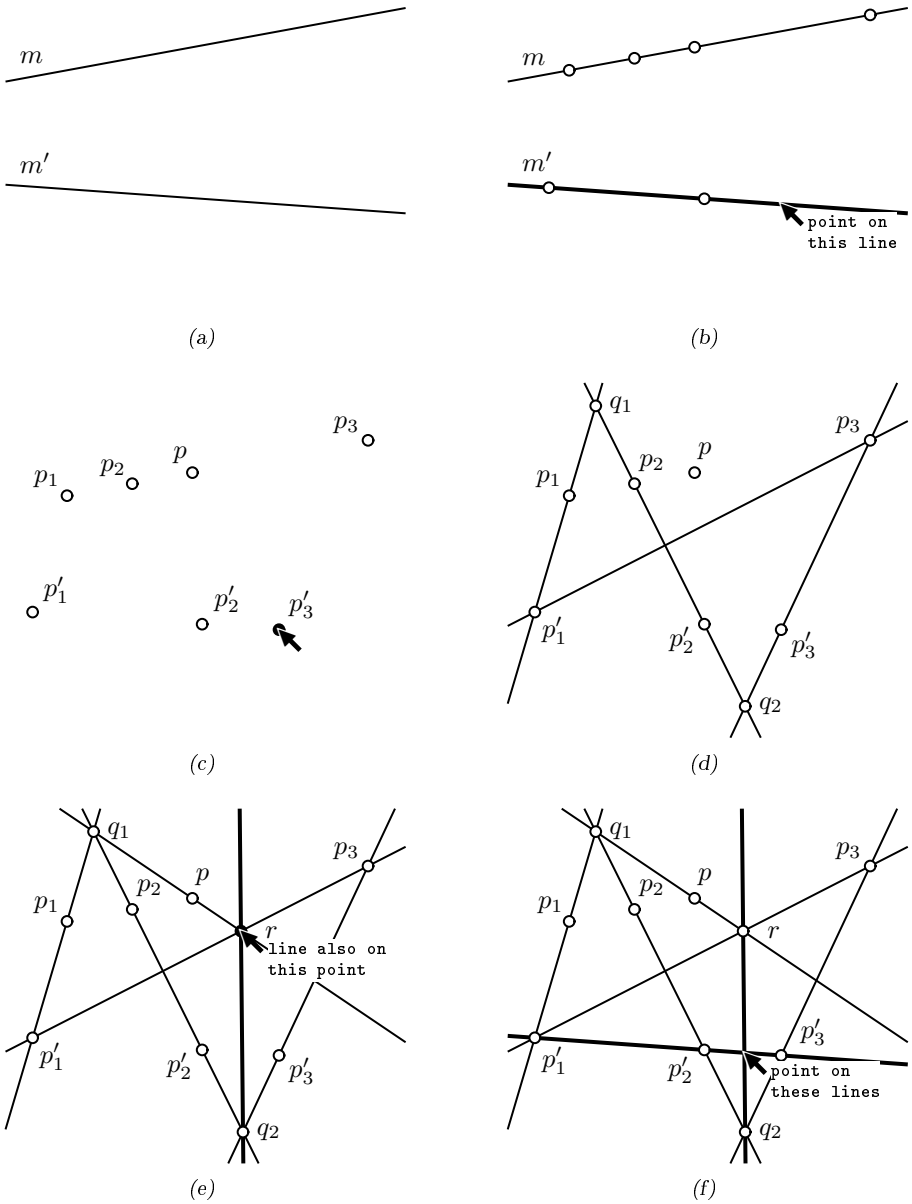
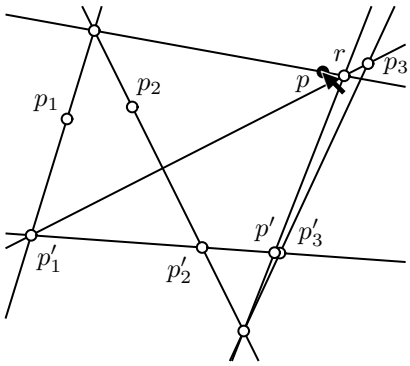
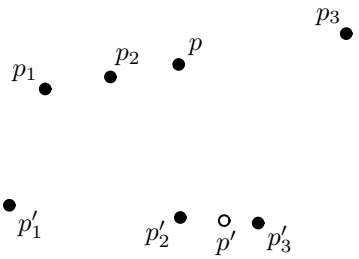


Figure 6.13. Constructing the P_1 *projectivity which maps three collinear points to three other collinear points.*



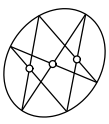
(g)



(h)

(i)

Figure 6.13. Continued.



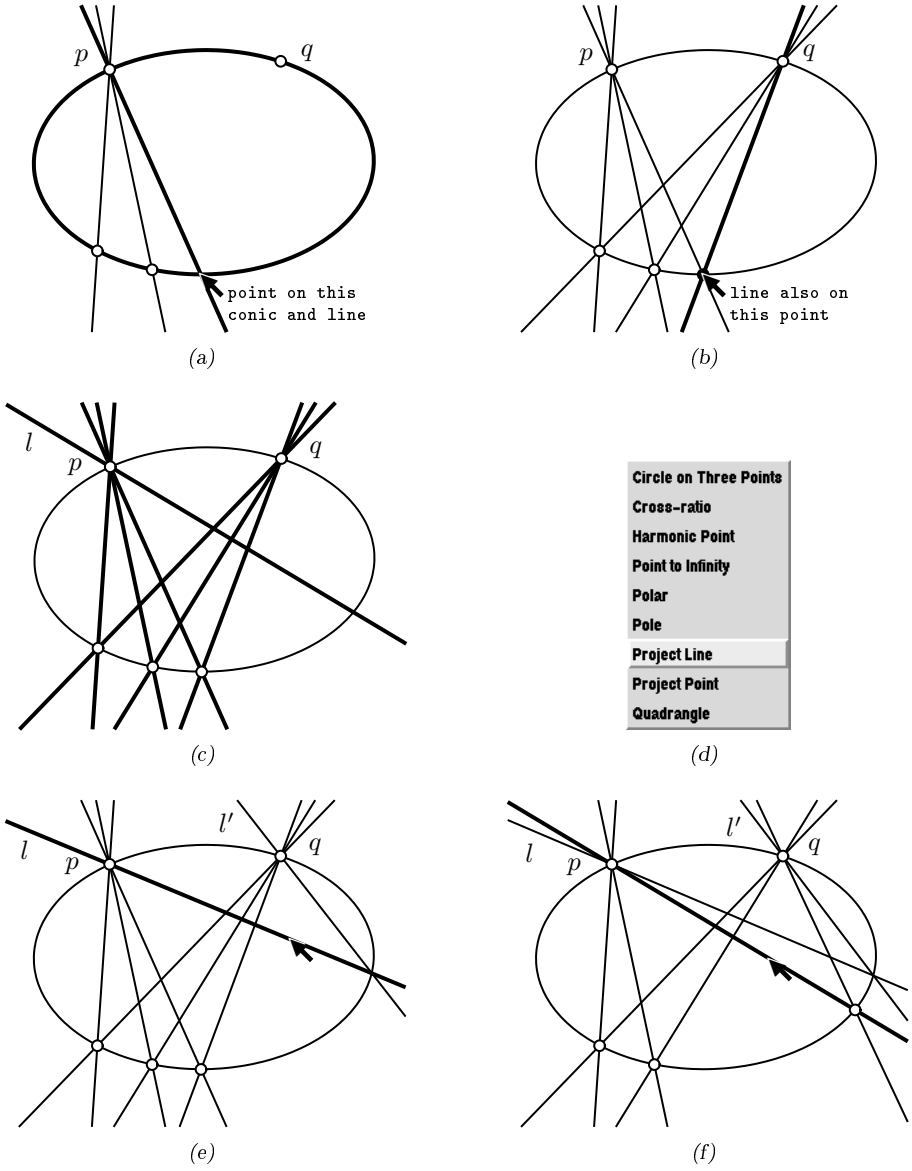


Figure 6.14. Constructing the projectivity which defines the conic.

intersect in a point on the conic. By dragging l, we can verify that this is true for all points on the conic.

The P_1 mapping is unique to the conic and will not be affected if we drag the lines on p. As shown in Figure 6.14f, the position of l' is not changed.

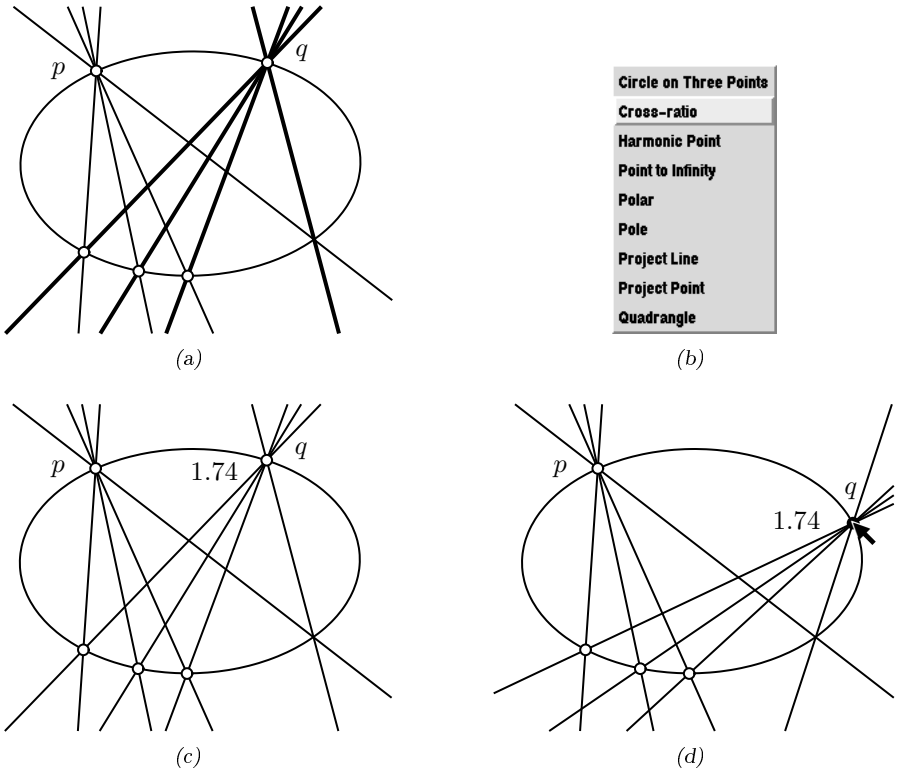
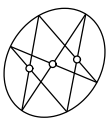


Figure 6.15. The cross-ratio of the four lines on q is independent of the position of q, as long as q is on the conic.

In Figure 6.14f, the cross-ratio of the four lines on p must equal the cross-ratio of the four corresponding lines on q. That follows from the fact that the two pencils are related by a P_1 projectivity. Furthermore, if q is dragged along the conic the cross-ratio of the four lines on q will be constant since the corresponding lines on p are fixed. That is easy to verify experimentally. Select the four lines and apply the `Cross-ratio` macro (Figures 6.15a-b). The cross-ratio is displayed close to the intersection point (Figure 6.15c). The value is not changed when q is dragged (Figure 6.15d).



6.2 Examples from affine geometry

6.2.1 Constructing the midpoint of a line segment

There is no metric in affine geometry. However, the existence of a line at infinity makes it possible to define some concepts that are usually associated with Euclidean geometry. For example, the midpoint of a line segment pq may be defined as the harmonic conjugate of the point at infinity on the line pq (Section 4.12.4).

Thus, we can construct midpoints using the harmonic conjugate macro that was created in Section 6.1.3. We will also use pdb's `Point to Infinity` macro which moves the selected points to the line at infinity. When applied to free points, the macro simply sets the homogeneous coordinate to zero. When applied to a point that has been attached to a line, the macro moves the point to the infinity point on that line.

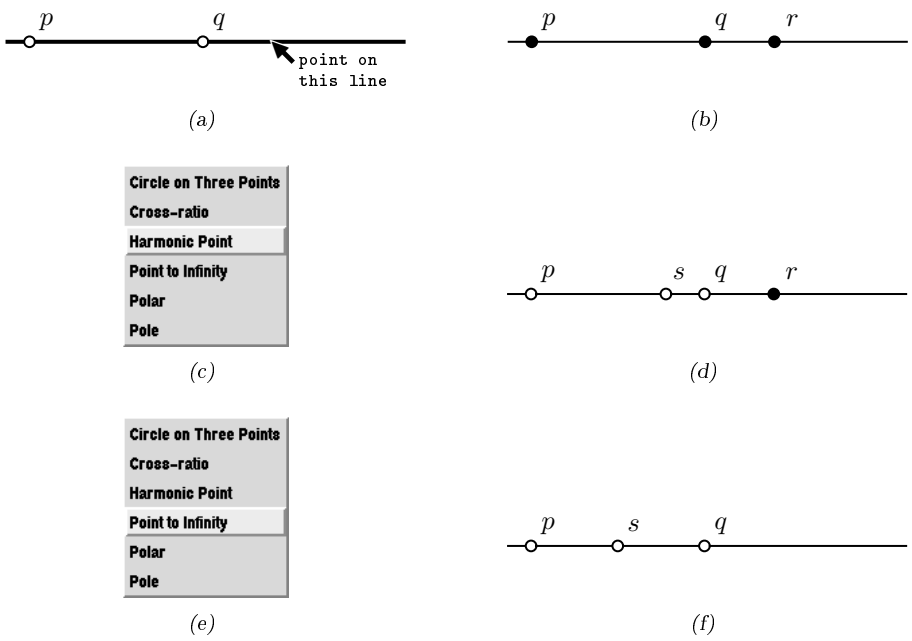


Figure 6.16. Constructing the midpoint of a line segment pq.

Figure 6.16a shows two given points p and q on a line. First, an auxiliary point r is placed on the line. Then we define s, the harmonic point of r with respect to p and q by selecting the three points and invoking the `Harmonic Point` macro (Figures 6.16b-c). Next, we select the point r and send it to infinity using the `Point to Infinity` macro (Figures 6.16d-e). As shown in Figure 6.16f, s becomes the midpoint of the segment pq.

6.2.2 Concurrent medians of a triangle

A line connecting the midpoint of the side of a triangle with the vertex that is opposite to the side is called a *median*, see Figure 6.17.

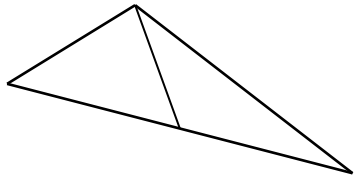


Figure 6.17. A median.

There is a theorem which states that the three medians of a triangle are concurrent. The theorem is often proved in elementary Euclidean geometry, but since *midpoint* is an affine concept, the theorem is really an affine one. Furthermore, it is a special case of the theorem discussed in Section 6.1.4 above.

Return for a moment to Figure 6.11g. To illustrate the affine version of the theorem, we need to place the line l at infinity. That can be done using the `Point to Infinity` macro in the following way. Create to auxiliary points and attach l to them (Figures 6.18a-c). Then select the auxiliary points and invoke `Point to Infinity` (Figures 6.18d-e). The result is shown in Figure 6.18f.

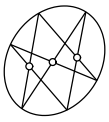
6.3 Examples from Euclidean geometry

6.3.1 The angle between two chords of a circle

It is a well-known Euclidean theorem that if q, r, s are three distinct points on a circle, the angle rqs is independent of the position of q. We can use pdb to illustrate this fact.

In Figure 6.19a, we have attached three points to a given circle. (The circle might have been created with the `Circle on Three Points` macro.) We have also defined the lines qr and qs. In Figures 6.19b-d, the (Euclidean) angle rqs is measured. The angle value will not change if q is dragged along the circle (Figure 6.19e).

This theorem can easily be proved using elementary angle formulas. Interestingly, however, the theorem follows immediately from Steiner's theorem and the discussion in Section 6.1.7. As explained in Section 4.12.3, the angle rqs is closely related to the cross-ratio of the four lines qr, qs, qI and qJ, where I and J are the circular points. Since every circle contains I and J, the cross-ratio of qr, qs, qI and qJ does not depend on the position of q (cf Figure 6.15d, page 190). Consequently, the angle rqs is constant.



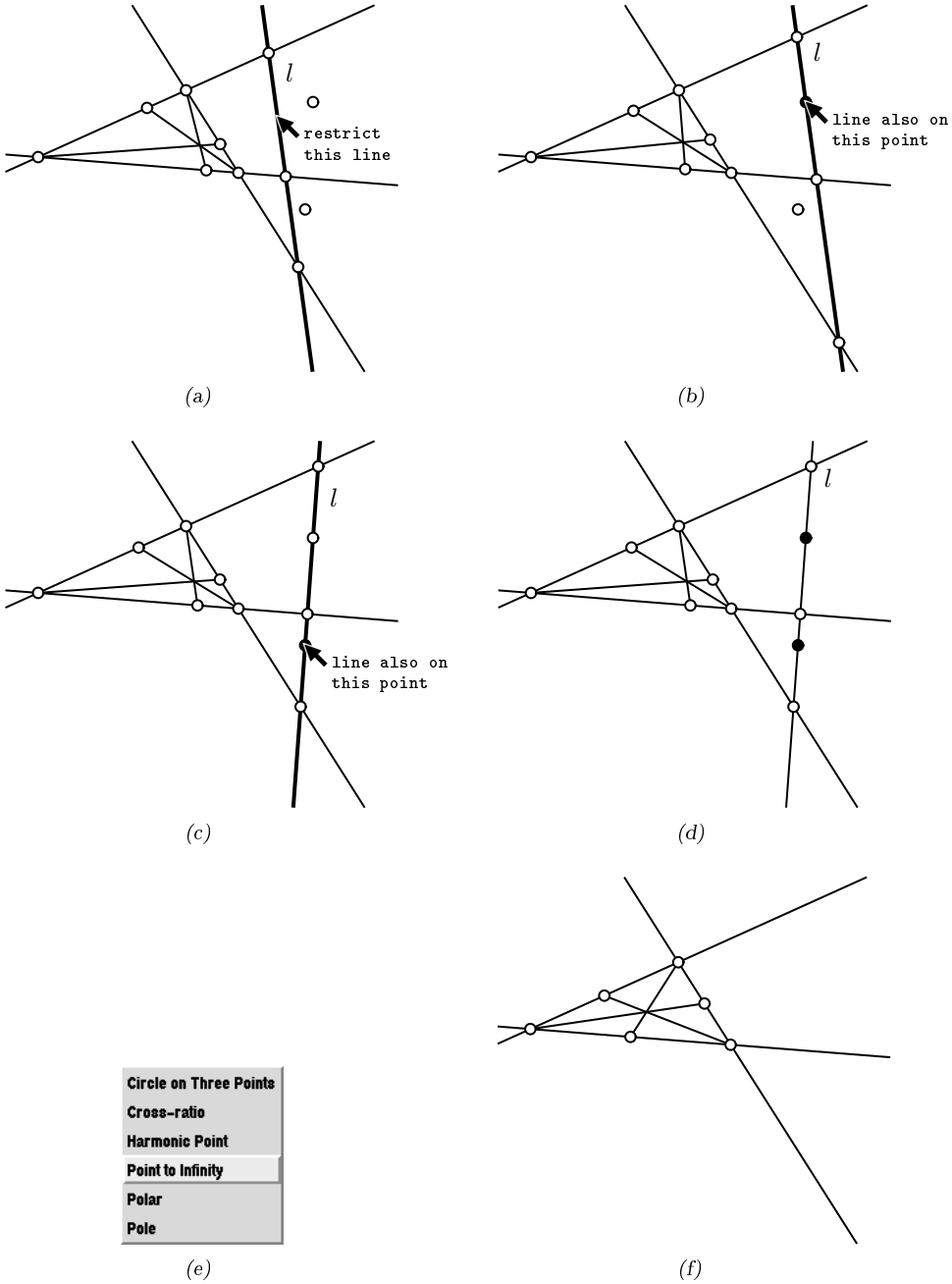


Figure 6.18. The three medians of a triangle are concurrent.

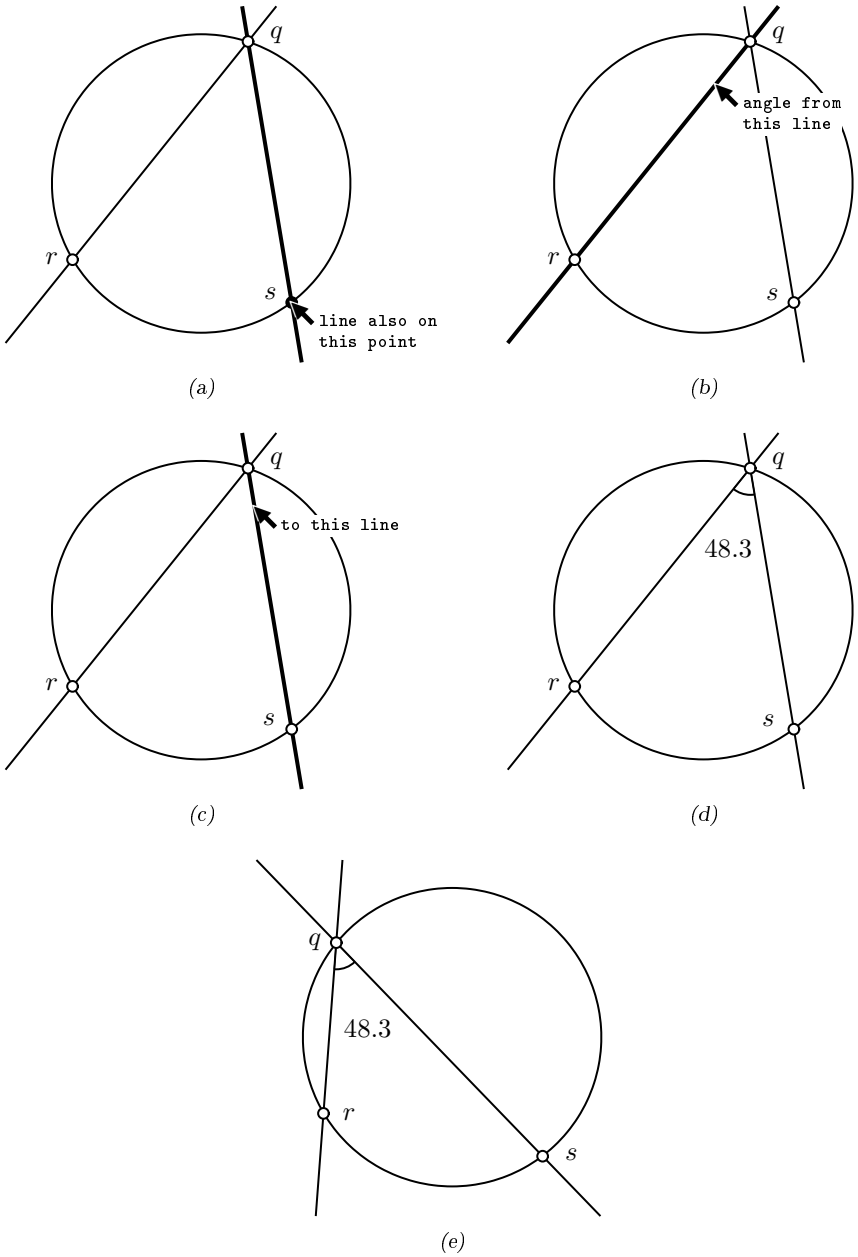
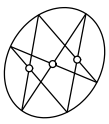


Figure 6.19. The angle rqs is independent of the position of q on the circle.



6.3.2 Confocal conics

In Section 4.12.3, we defined the focal points of a conic as the points where the ideal tangents of the conic intersect each other. It follows that confocal conics share the same four ideal tangents, see Figure 4.42 on page 73. In this section, we will show how to generate a set of confocal conics in pdb. To make the construction easier to follow, we will use two ordinary points instead of the absolute points I and J (as in Figure 4.42). Later, we will repeat the same construction starting from the two absolute points.

The goal is to construct a conic which has the same focal points as the conic given in Figure 6.20a. The first step is to place two arbitrary points (which will play the roles of I and J) on the background, and draw the four tangent lines. A fifth, free line is also placed on the background (Figure 6.20b). By activating the conic tool and picking the five lines, we can create a conic that is tangent to the five lines (Figure 6.20c). To record this construction as a macro, we select the original conic, the free line and the two points, then invoke `Edit`→`Create Macro` (Figures 6.20d-e). We will name this macro `Confocal`.

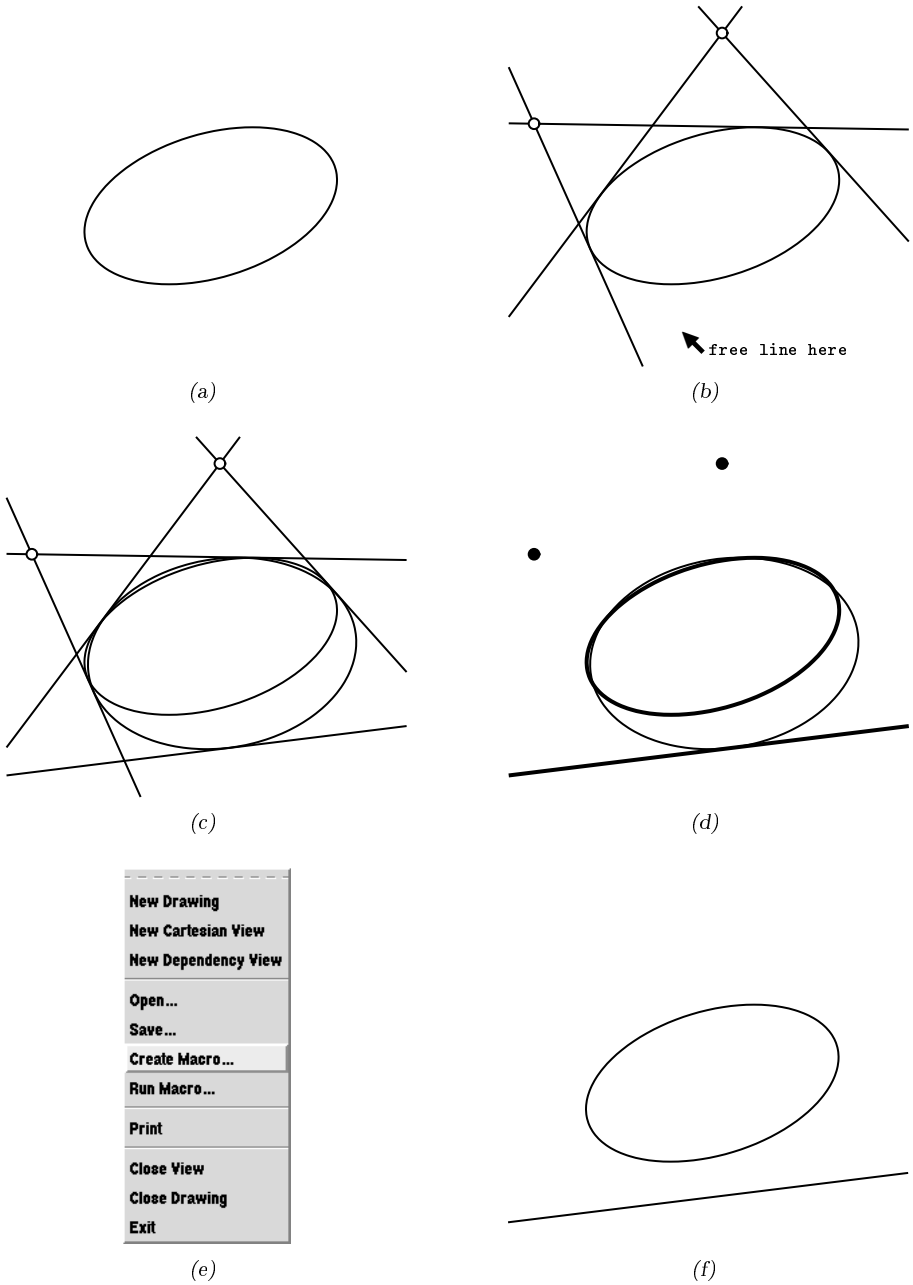
To obtain a confocal conic we will now apply the `Confocal` macro to the given conic, the free line, and the absolute points I and J. The only problem is how to refer to I and J? The most straightforward method is probably to enter the (complex) coordinates of I and J from the keyboard. However, we will show a trick that saves us from typing. We apply the standard `Circle on Three Points` macro[1] to three arbitrary points (Figure 6.20g). The circle has five parents: I, J and the three points shown in (g). By selecting the circle and invoking `Edit`→`Select Parents`, we place the parents of the circle in the current selection. Then we deselect the three ordinary parent points by enclosing them while holding the `Shift` key down (Figure 6.20i). The current selection now contains only I and J. We add the conic and the free line to the section and invoke the `Confocal` macro we prepared above (Figure 6.20j). The result is shown in Figure 6.20k. By dragging the free line, we can pick any conic in this set of confocal conics. The hyperbola in Figure 6.20l is one of them.

How can we create the focal points? The `Confocal` macro did not display the ideal tangents in Figure 6.20k because the tangents were not visible when the macro was recorded. However, it is easy to make them visible. We just select the original conic and invoke `Edit`→`Select Children` followed by `Edit`→`Create Images`. The ideal tangents are displayed in gray since their coordinates are complex (Figure 6.20p). Nevertheless, with the point tool active, we can still place points on their intersections (Figure 6.20q). As explained in Section 4.12.3, two of them will be real. They are the ones we usually call focal points.

6.3.3 A theorem of Poncelét

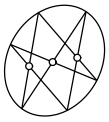
We can use the focal points from the previous section to illustrate the following theorem due to Poncelét. Place an arbitrary point p on the drawing board. Draw

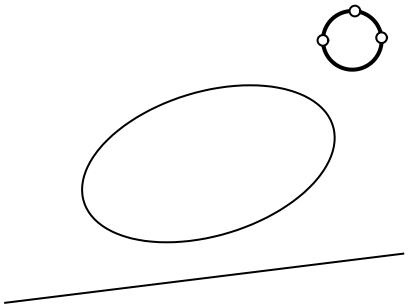
[1] The macro was discussed in Section 5.3.6.



(a) (b) (c) (d) (e) (f)

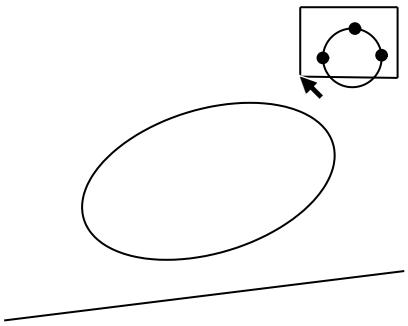
Figure 6.20. Constructing confocal conics.



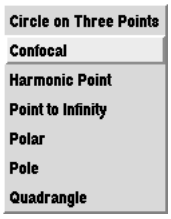


(g)

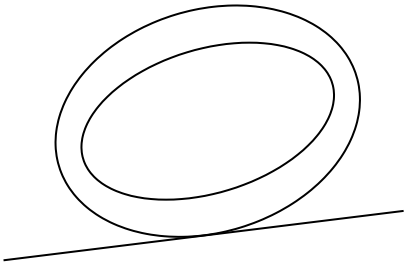
(h)



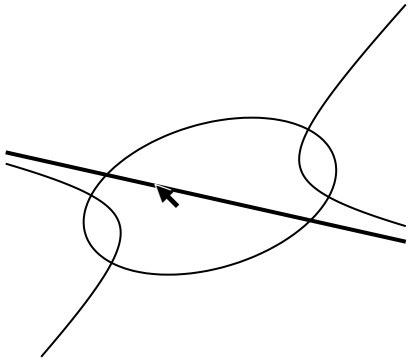
(i)



(j)

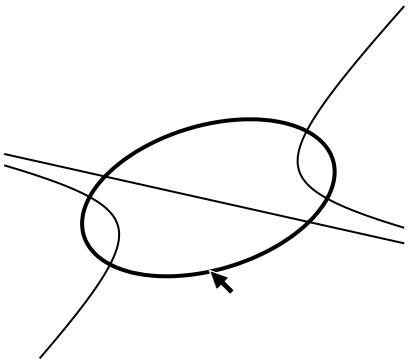


(k)



(l)

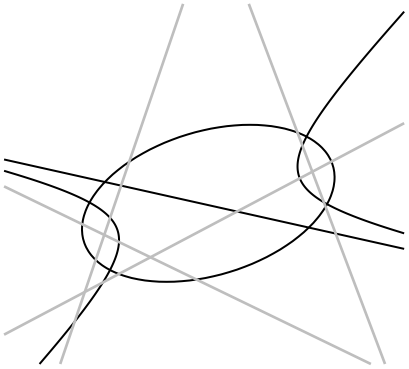
Figure 6.20. Continued.



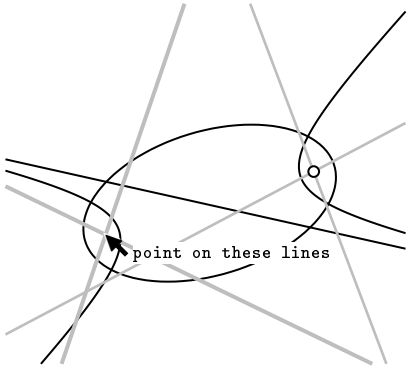
(m)

(n)

(o)

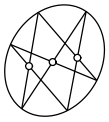


(p)



(q)

Figure 6.20. Continued.



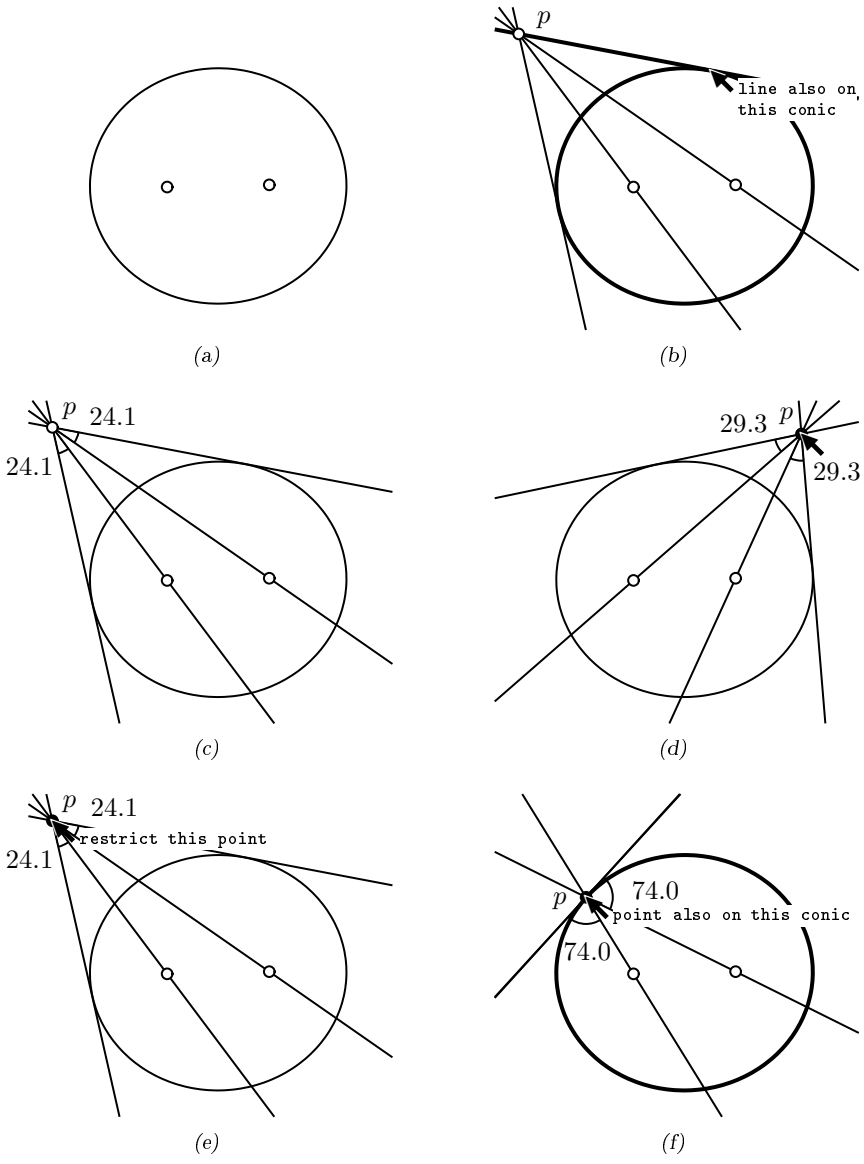


Figure 6.21. A theorem of Poncelét.

the tangents through p and the lines which connect p with the two focal points (Figures 6.21a-b). It turns out that the two angles that we have measured in Figures 6.21c-d are always equal. If p is attached to the conic, the theorem specializes to the reflection law (Figures 6.21e-f).

6.4 Examples from hyperbolic geometry

6.4.1 Constructing an isometry

In a non-degenerated geometry, such as hyperbolic geometry, angles and distances can be defined in terms of a proper, absolute conic. The isometries are exactly those projectivities which leaves the absolute conic invariant (Section 4.11.5). Here, we will construct such a projectivity. Furthermore, we will construct it in such a way that we can obtain *all* isometries just by varying the position of a few points (see also [Coxeter98]).

We showed in Section 4.8.12 that a projectivity which preserves a conic is completely determined by three pairs of corresponding points on the conic. Given an absolute conic Ω and six points $p_1, p_2, p_3, p'_1, p'_2, p'_3$ on Ω (Figure 6.22a), we will construct a projectivity on P_2 which maps Ω onto itself and which takes p_1 to p'_1, p_2 to p'_2 and p_3 to p'_3.

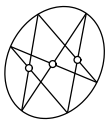
We start by mapping points on Ω. Then we will extend that map to the rest of the projective plane. In Figure 6.22b, the *Pascal line* corresponding to $p_1, p_2, p_3, p'_1, p'_2, p'_3$ has been drawn. Given an arbitrary point p on Ω, we draw the line pp'_2. If that line intersects the Pascal line in r, we draw rp_2 and define p', the image of p, as the intersection of rp_2 and Ω (Figure 6.22c). By dragging p along Ω we can verify that this projection actually maps p_1, p_2, p_3 to p'_1, p'_2, p'_3 (Figure 6.22d). Actually, the fact that $p_3 \mapsto p'_3$ is a consequence of Pascal's theorem.

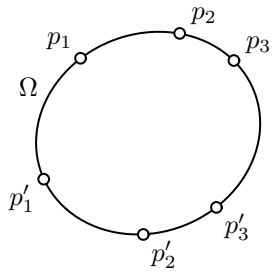
Since we will need to map several points, we save this construction as a macro. We select the conic, $p_1, p_2, p_3, p'_1, p'_2, p'_3$ and p, then invoke `File→Create Macro` (Figure 6.22e).

Now, given an arbitrary point q_1, not necessarily on the conic, we draw the tangents through q_1 and define the tangent points t_1 and t_2. We can obtain the images of the tangent points (t'_1 and t'_2) using the macro we defined above (Figure 6.22f). The tangents through t'_1 and t'_2 intersect in q'_1 (Figure 6.22g). Since a projectivity preserves incidences, q'_1 must be the image of q_1. We now have a projection on P_2 which preserves Ω, i.e., an isometry.

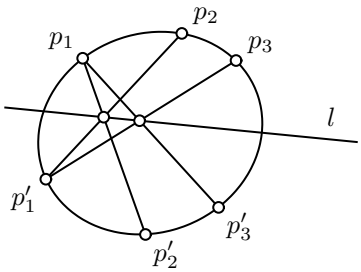
We save the construction as a macro and apply it to a second point q_2 (Figures 6.22h-i). To verify that the projectivity really is an isometry, we can compare the distances q_1q_2 and $q'_1q'_2$. Of course, we should use the metric defined by Ω. As shown in Figure 6.22j, the two distances are equal. By using a third pair q_3, q'_3 of corresponding points, we can in a similar way confirm that the angles $q_1q_2q_3$ and $q'_1q'_2q'_3$ are equal.

By dragging any of the points $p_1, p_2, p_3, p'_1, p'_2, p'_3$, we obtain other isometries.

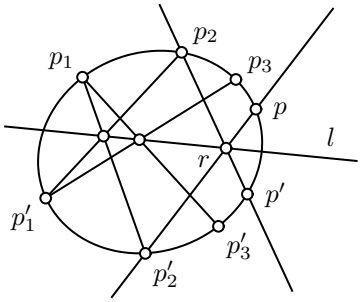




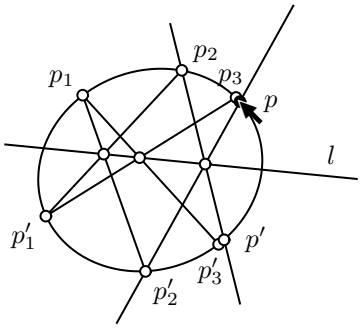
(a) A projectivity T which leaves a conic invariant is determined by three pairs of corresponding points on the conic.



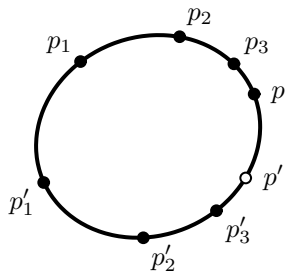
(b) The Pascal line l.



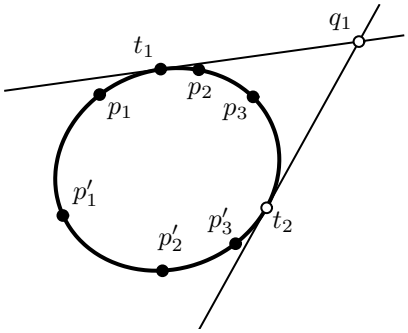
(c) Mapping an arbitrary point p on the conic to $p' = Tp$.



(d) Verifying visually that this construction actually maps p_1, p_2, p_3 onto p_1', p_2', p_3'.

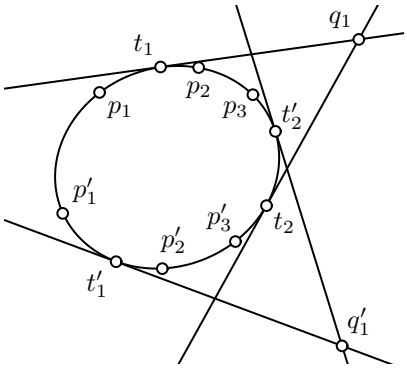


(e) Selecting the macro arguments just before creating a macro which maps points on the conic.

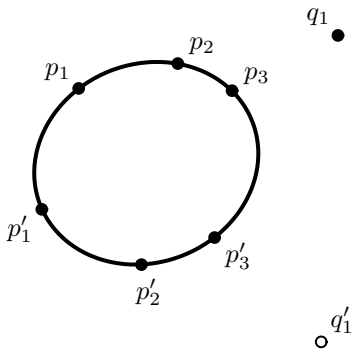


(f) Selecting the macro arguments just before applying the macro to create the image of t_1.

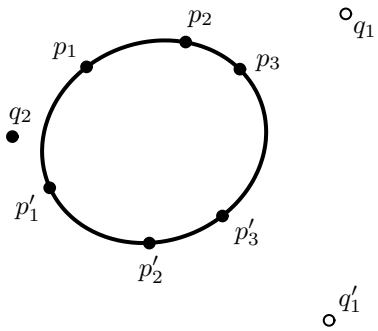
Figure 6.22. Investigating isometries in hyperbolic geometry.



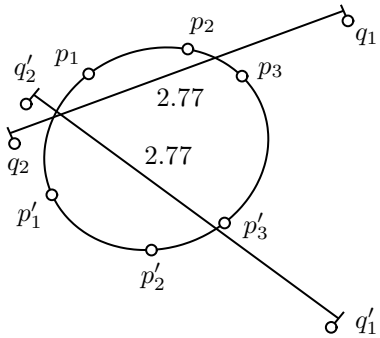
(g) The image of q_1 is constructed as the intersection of the tangents through t'_1 and t'_2.



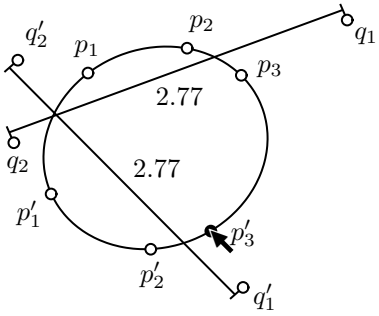
(h) Selecting the macro arguments just before creating a macro which maps an arbitrary point in the plane.



(i) Selecting the macro arguments just before applying the second macro to create the image of q_2.

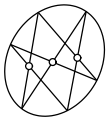


(j) Comparing $\operatorname{dist} q_1 q_2$ and $\operatorname{dist} q'_1 q'_2$ in a metric in which the conic is the absolute.

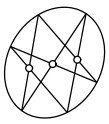


(k) Modifying the mapping T by dragging p'_3.

Figure 6.22. Continued.



As long as q_1, q_2 and the conic are fixed, the distance between q_1 and q_2 and between q_1' and q_2' will be constant (Figure 6.22k).



Chapter 7

Conclusions and generalizations

How well does pdb compare with other, similar systems? What lessons can be learned from its design? What is the future of dynamic geometry? We will start by comparing pdb with two other tools in the next section. In Section 7.2, we evaluate two decisions that had a large impact on the design of pdb. The need for evaluating pdb in real classroom situations is pointed out in Section 7.3. Finally, in Section 7.4, we discuss how the current system could be developed further.

7.1 Comparing pdb with other, similar systems

In this section, we compare pdb with two other, similar systems: Cabri (cf Section 2.1) and Cinderella Café (cf Section 2.3). We will describe how pdb differs from these systems and in what way pdb constitutes an improvement.

The reason we have chosen Cabri and Cinderella Café for comparison is that they take the same construction-oriented approach to dynamic geometry as pdb (cf Section 5.1.3), and that all three systems have similar application domains and potential user groups. Cabri is much more widely used than Cinderella Café, but both systems must be considered to represent state-of-the-art in dynamic modeling.

Our comparison is based on the program versions and written documentation that are available at the time of writing (January 1999).

7.1.1 Expressive power

Support for simple incidence constraints between points and lines is all that is needed to study, for example, the properties of quadrangles and quadrilaterals (Section 4.10) or to illustrate the theorem of Pappus (Section 1.3). However, to illustrate e.g. Pascal's theorem we also need conics. To study the relationship

between a quadrilateral and its inscribed conics, we must be able to define tangency constraints. To experiment with the focal points and the axes of a conic, it must be possible to define the circular points (Section 4.12.3), and so on. Thus, the more expressive power a system has, the more advanced are the drawings it can produce. Let us compare pdb, Cabri and Cinderella Café in this respect.

- pdb supports several types of tangency constraints. If a free line is attached to a conic, it becomes a tangent to that conic. The line will have one degree of freedom left and can therefore be dragged along the conic. If the line is also restricted to be incident with a point, it can occupy two different positions (see Figure 5.1, page 81). pdb makes sure that the line does not jump between these positions if either the conic or the point is moved. However, pdb allows the user to *drag* the line between the two allowable positions.

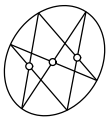
  Cabri has no primitive for representing tangent lines. A line that is tangent to a conic C and also incident with a given point p must be created using, for example, the construction shown in Figure 5.15, page 97. However, such a construction is sensitive to the placement of the auxiliary points. As a result, the tangent line will jump unpredictably if the conic is moved. Moreover, since Cabri does not support complex coordinates, the construction breaks down if the point p is moved inside the conic.

  The currently available version of Cinderella Café draws the tangents through a given point p as a line pair, i.e., as a degenerated conic. It it not possible to show just one of these lines. The two lines are not distinguished, so a point placed on one of the lines may very well slide over to the other line if the conic is moved. (This will probably be fixed in the next release of Cinderella Café.) Like Cabri, Cinderella Café cannot handle complex tangents.

  Moreover, in contrast to pdb, neither Cabri nor Cinderella Café allows the user to draw the common tangents of two conics. Consequently, they cannot, for example, represent the drawing shown in Figure 6.12, page 184.

- As already mentioned, pdb can perform complex arithmetic and represent complex points and lines on the screen (Section 5.3.4). This capability, which both Cabri and Cinderella Café lack, makes many constructions involving conics more general and useful. For example, the polar construction shown in Figure 5.47 on page 144 works even if the point is inside the conic. Furthermore, having complex coordinates allows the users of pdb to use the circular points I and J (Section 4.12.3) for creating circles (Figure 5.55, page 153), focal points (Figure 6.21, page 199), confocal conics (Figure 6.20, page 196), etc. Support for complex geometry also makes it possible to use an elliptic metric, which is defined by an absolute point conic with a complex point set (Section 4.12.2).

- In Cabri, all distances and angles are assumed to be Euclidean. In contrast, in a pdb drawing, the metric can be specified for each single measurement



and metric constraint. The currently available version of Cinderella Café does not support measurements or metric constraints.

- pdb is the only system which allows the user to open several views whose coordinate systems are related by general projectivities[1]. Thus, a point which is at infinity in one view might be a perfectly ordinary point in another view. This feature is very useful since it allows the user to interact with objects at infinity and see the effect on dependent objects closer to the origin. However, as explained in Section 5.3.3, it also means that, in general, there will be no one-to-one correspondence between the angles (or distances) in different views. Thus, measurements and metric constraints become view-dependent. pdb has been designed to cope with this view-dependency and to handle measurements and metric constraints consistently in all situations. For example, if the user adds a equality constraint $\alpha = \beta$ between two angles (e.g. by dragging α onto β), pdb first makes sure that α and β have been measured in the same metric so that they are comparable, then records that the constraint should hold in that particular metric. Furthermore, pdb chooses a suitable default metric for each view so that, for example, the angle between two lines that appear to be perpendicular on the screen will have the value $\pi/2$. The default metric chosen by pdb depends on the type of view: a `CartesianView` (Section 5.3.1) is associated with a Euclidean metric which has the same line at infinity as the view, and a `PoincareView` is by default associated with a hyperbolic metric whose absolute conic is projected onto the rim of the Poincaré disc. However, a view can be explicitly associated with any type of metric. Furthermore, any conic in the drawing can serve as the absolute conic for a (non-degenerated) metric. This feature was used in Figure 6.22 on page 201. By changing the shape of the absolute conic, the metric can be modified dynamically. The support for defining and using metrics found in pdb is to our knowledge unmatched by other tools.

Thus, compared to existing systems, pdb contains several new features which allow us, for the first time, to visualize a number of interesting geometric constructions in dynamic, interactive drawings.

7.1.2 User interface

In Section 5.2.3, we discussed a number of interaction problems that can be found in virtually every other dynamic geometry system. Particularly annoying are what we termed *jumping* and *drifting* objects since they often cause a geometric configuration to collapse when the user moves an object slightly. For example, the undesirable effect illustrated in Figures 5.18 and 5.19 on page 102 can be seen in Cabri drawings. Moreover, Cabri does not always distinguish an angle from its complement. Consequently, an angle α in a Cabri drawing can suddenly take the value $\pi - \alpha$. If α is used in metric constraints, all objects whose positions are

[1] Actually, this feature was introduced in drawing board system discussed in [Naeve89].

determined by α will jump at the same time. Drifting objects (Figures 5.16 and 5.17, page 100) is a major problem in Cinderella Café.

Two basic requirements, *continuity* and *repeatability*, which every dynamic geometry tool should fulfill were formulated in Section 5.2.4. In Sections 5.2.5 through 5.2.7 we showed that these requirements could be met in most situations using oriented geometry for representing the position of objects, signatures for identifying the roots of equations, and P_1 coordinates (cross-ratios) for representing angles and distances. The methods we suggest have all been implemented and tested in pdb. Combined, these improvements give the user interface of pdb a stability and smoothness not provided by any other comparable tool for dynamic geometry currently available.

The drag-and-drop operations supported by pdb resemble those of Cabri (and other tools, e.g. GSP, see Section 2.2). In contrast to Cabri, however, pdb also allows all types of incidence constraints to be removed or redefined using the same drag-and-drop interface. This important feature allows the user to correct mistakes made earlier in a construction. It also allows the user to see why certain constraints are necessary. For example, the user could tear one of the points from the conic in Figure 1.4b on page 6 to verify that Pascal's theorem is valid only if all six points are on the conic. Finally, only pdb allows angles and distances to be copied using simple drag-and-drop operations.

Apart from this, there are three other noteworthy improvements to the user interface compared with Cabri and Cinderella Café.

- In contrast to the macro facilities in other tools, pdb's macro language (Tcl) gives the user full access to the entire system. It is possible to define a drawing, to alter an existing drawing, to create animations and to perform advanced matrix computations in a Tcl script.

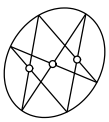
- pdb is the only system which can show the structure of a drawing (i.e. the dependencies among the objects) as a graph (Section 5.3.1). The user can easily modify the graph should he not be satisfied with the default layout. It is easy to see the correspondence between the dependency graph and the other views since the objects are consistently named and colored in all views. Also, when the user selects an object, it is highlighted in all views simultaneously.

  Cinderella Café can list the dependencies in text form, much like a macro, but cannot display them graphically. Cabri has no such feature at all.

- pdb can produce high-quality PostScript output, which can be scaled and inserted into a written document. In fact, most figures in this thesis have been produced with pdb.

7.1.3 Performance

As mentioned in Section 5.4.1, we consider a short response time to be one of the most important design criteria. If the response time is too long, the objects on



the screen will lag behind the cursor during dragging operations. The drawing will then feel jelly-like and will be hard to control. The importance of having almost instantaneous, accurate feedback cannot be emphasized enough.

We think that the response time of pdb is satisfactory, and significantly shorter than existing tools, especially for complicated drawings. However, it is difficult to make an accurate and fair comparison of response times, for three reasons:

- Some tools, such as pdb and Cabri, have been compiled into native machine code. In contrast, Cinderella Café runs on the Java virtual machine, which induces a significant run-time overhead. This overhead will be reduced if Cinderella Café is compiled into native code. Thus, comparing pdb and Cabri with the interpreted version of Cinderella Café is not very meaningful. It is reasonable to assume, however, that Cinderella Café, like most applications written in Java, must scarify some of its speed for increased flexibility and portability.

- The three systems currently run in completely different environments. For example, Cabri is only available for Microsoft Windows and for the Macintosh. On both systems, Cabri uses a native and very fast windowing system. In contrast, pdb runs under UNIX and uses XWindows, a network-based windowing system. That will of course reduce the speed at which the contents of the windows can be updated, especially if pdb and the windowing system server communicates over a network. On the other hand, a standard PC is still significantly slower than the average UNIX workstation. Cinderella Café uses the windowing capabilities provided by the Java environment. Consequently, if Cinderella Café is executed from within a web browser, the characteristics and quality of the browser will affect the overall performance.

- Depending on the set of geometrical primitives available in each tool, certain types of drawings will be updated faster than others. This is the classical problem with performance evaluations: we have to agree on a set of representative benchmarks.

Because of this, we have not attempted to measure and compare the response time of the tools. However, such a comparison would definitely be valuable if properly conducted. A Microsoft Windows version of pdb is planned, and could be compared directly with the Windows version of Cabri. The next version of Cinderella Café will probably be available as native code for at least some UNIX system, and could then be compared with the UNIX version of pdb.

It must be understood, though, that there is a conflict between keeping the response time short and having a smooth and stable interface. For example, to avoid the problems described in Section 5.2.3, a significant amount of computation is required each time the drawing is updated. Thus, in order to make a fair comparisons of the response time of different systems, the overall impression of the user interface must be taken into account.

7.2 Two major design decisions in retrospect

7.2.1 Internal representation

As explained in Chapter 2, dynamic geometry systems fall roughly into two categories, constructive and constraint-based. pdb, Cabri and Cinderella Café all belong to the first category. With these tools, a drawing is defined as a sequence of constructional steps. In contrast, a constraint-based system can create a drawing from a given set of constraints. The pros and cons of the two approaches were discussed in Section 5.1.4.

While developing pdb, we have found it very important to have access to the simple and explicit representation of the geometric constraints that the dependency graph provides. The graph made it possible to speed up computations and to give the user appropriate feedback in every situation.

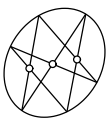
However, if more types of objects and/or constraints are added to the system, the number of different node types needed in the dependency graph grows very fast. Up to, say, 50-100 different combinations of objects and constraints, the situation still remains manageable. Above that limit, the dependency graph gets too complicated, and a more uniform, algebraic representation might be more appropriate.

Nevertheless, we strongly believe that choosing a constructive representation for the current version of pdb was the correct decision as it allowed us to improve the user interface significantly while keeping the response time down. In that respect, pdb has set a new standard for the user interfaces of dynamic geometry systems. If we choose a different internal representation for future versions of pdb, we must make sure that it has no negative effects on the user interface. There are several examples of constraint-based geometry systems whose internal representations, although simple and elegant, require heavy computations and still fail to support important aspects of the user interface. The result has been slow systems which are hard to use.

Curiously, most constraint-based geometry systems seem to be based on logic programming and several systems have actually been implemented in Prolog. That seems strange since the geometric constraints give rise to non-linear systems of polynomial equations which have very little to do with predicate logic. Not surprisingly, these systems have problems even with the simplest of incidence constraints, such as tangencies between lines and conics. We think that systems of polynomial equations should be solved using general and powerful algebraic methods, such as Gröbner base techniques (Section 5.1). Only with the help of such a method will the software have a chance to analyze the geometric constraints and guide the user through the construction.

7.2.2 Implementation language

For reasons discussed in Section 5.4.3, we chose C++ as the primary implementation language for the pdb kernel. The main reason was that we anticipated



a need for extensive code optimization in order to achieve an acceptable performance for complicated drawings. And indeed, it did turn out to be necessary to use e.g. in-line code expansion, loop unrolling, and fast iterators for accessing the contents of containers. C++ has been designed to allow for that type of low-level optimization, and we do not believe that a comparable speed-up could have been achieved with a language such as e.g. Java.

C++ has a reputation for being overly complicated and hard to use. However, with the standardization of the language and its library, the situation has improved dramatically. In fact, the C++ language itself has not caused us any problems. Instead, the main difficulty has been that we also have had to use a second, interpreted language for certain tasks. As explained in Section 5.4.3, it is important that the user has access to a macro language which does not require compilation or linking. We chose to extend the Tcl language and to define an interface between Tcl and C++ using TIDE (see Appendix A). The use of two different languages in an object-oriented system raises several questions. For example, in what language should the objects be implemented and what should the division of responsibilities be? If we would have used Java for implementing the pdb core, there would have been less need for a second, special macro language, since Java can also run interpreted. Thus, a single language might have sufficed, which would have simplified the implementation considerably. However, the performance of the system is much more important to the end-user than how the software is organized internally, and in that respect we still think that the C++/Tcl solution was the best choice.

7.3 Evaluating the tool in real teaching situations

So far, pdb has been tested only by a small group of people. A few live demonstrations have also been given in different contexts. What is obviously lacking is a proper evaluation of pdb in a realistic teaching situation. A number of informal, unpublished tests in classroom situations were performed with a forerunner of pdb back in 1988 [Naeve89]. Some of observations that were made then have influenced the design of pdb, e.g. the need for having the dependency graph displayed on the screen in order to understand the structure of a given drawing. It is now time to put pdb to a similar test. There are a number of questions that need to be answered.

- In what ways can the use of computerized visualization tools, in particular dynamic geometry systems, make it easier to teach geometry? The new generation of students will be used to computer-generated animations and highly interactive computer games. Therefore, we believe that the ability to experiment, in real-time, with complex geometric constructions will appeal to them and encourage them in their further studies, not only in geometry, but in mathematics in general. It is however an open question how the use of computer-based visualization and experimentation tools is best integrated

into the mathematical curriculum. There are surely pedagogical pitfalls that can only be discovered through experimentation [Wilson99].

- For what forms of teaching is pdb best suited? A dynamic geometry system can be used in several ways. In a traditional course, based on a textbook, the students might use the system only for certain exercises. During lectures, the teacher could use the system as an alternative to drawing on a whiteboard. In fact, we are currently experimenting with using pdb on a touch-sensitive whiteboard, which allows the teacher to create and manipulate dynamic drawings, using only the tip of his finger. One could also imagine a web-based course centered around an on-line textbook. Such a textbook could be illustrated with dynamic drawings with which the students could interact. It would not be too difficult to extend pdb with a multi-user capability which would allow the teacher to help the students with exercises and assignments remotely over a network (Section 7.4.2).

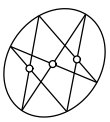
- How can the unique features of pdb, such as its ability to handle objects in the complex projective plane and its extensive support for handling metrics, be used to better illustrate the relationship between different geometries?

- How useful is pdb (and dynamic geometry systems in general) for exploring the geometric structure of problems in e.g. robotics and computational vision?

- What are the users' impressions of the graphical interface of pdb? In particular, are the drag-and-drop operations for adding and modifying constraints intuitive? Is the textual feedback provided by the pdb drawing tools relevant? Is the dependency graph useful for understanding how a construction works and for identifying objects?

However, such a study must be carefully prepared. No serious conclusions could be drawn from a study where pdb, or another tool for dynamic geometry, is put into the hands of unprepared students. One of the main ideas behind the pdb project was to build a drawing tool based on purely projective concepts, but with the capability of handling different types of metrics. We believe that such a tool could, if used properly, help to illustrate and to clarify the relationship between projective, Euclidean and non-Euclidean geometry. Some examples along these lines were given in Chapter 6. It is unlikely, though, that such examples would be appreciated unless the students are familiar with the basic concepts of projective geometry.

For a first evaluation, we therefore envisage a short course in projective, Euclidean and non-Euclidean geometry based on a good, standard textbook, where the exercises are carried out by the students using pdb. There would be traditional, live lectures where the teacher could use pdb e.g. on a touch-sensitive whiteboard. Supplementary material, illustrated with pdb drawings would be available on the web.



To create such a course, to carry it out successfully, to make observations and to draw conclusions would be a major undertaking. It would be necessary to establish a good working relationship with the teachers and to sell the idea of reintroducing a serious geometry course into the curriculum. A rough time estimate for all of this is at least one man-year. In fact, there would be enough work for another thesis. The focus would then be on the general didactic questions related to computer-aided mathematical education, and not so much on the capability and the user interface of the geometry system itself.

To summarize, we have so far concentrated entirely on designing and building the software, in the firm belief that it is not meaningful to do half-hearted evaluations of poorly designed tools or unstable implementations. However, we have now reached a point where we are ready to evaluate pdb in a classroom situation, to investigate what the benefits of computer assistance in teaching are and, more specifically, evaluate how well pdb compares to other, similar tools in terms of performance and user friendliness. In fact, the Centre for User Oriented IT Design (CID) at KTH is currently applying for funding to carry out such a project in cooperation with the Department of Teacher Training (ILU) at Uppsala University.

7.4 Future development

We regard the development of pdb as a first step towards building a more general environment for working with dynamic geometry. Several improvements and generalizations of pdb are possible. Some of these improvements fit well into the current framework and would therefore be relatively easy to add. Two such examples are discussed in Section 7.4.1. In contrast, the extensions discussed in Section 7.4.2 would require major changes to the software.

7.4.1 Minor improvements of the current system

Degenerated conics

Currently, all conics in pdb are assumed to be proper. However, allowing conics to degenerate would be quite useful in certain situations, e.g. for showing how a non-Euclidean geometry is transformed into Euclidean geometry when the absolute conic collapses.

It would not be difficult to check if the coefficient matrix of a conic is close to being rank deficient and in that case, draw the conic as a pair of lines. However, since there is no 1-1 correspondence between degenerated point conics and degenerated line conics (cf Section 4.8.13), we would also have to make a distinction between point conics and line conics in general. That would have consequences also for other objects. For example, a metric (Section 4.11.1) would not be defined by a single absolute conic but rather by a pair of possibly degenerated conics: a point conic Ω for measuring distances and a line conic Ψ for measuring angles. Thus, allowing for degenerated conics requires some additional work.

Tangency constraints on conics

Currently, it is not possible to require two conics to be tangent in a point. Such tangency constraints are interesting e.g. when studying hyperbolic geometry where circles are conics that have double contact with the absolute (Section 4.11.4).

7.4.2 Major extensions

Multi-user capabilities

pdb is currently a single user system. However, when used for collaborative work, pdb would have to have some sort of multi-user capability. For example, students taking a network-based distance course in geometry would typically be geographically spread out. While doing their assignments or exercises, they could ask a teacher working on a remote machine for help. It should then be possible for the teacher to attach to the student's pdb process and look at the student's drawing. If the teacher modifies the drawing, the student should be able to see the cursor movements and every operation that the teacher performs, including menu selections etc.

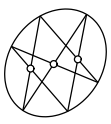
Obviously, the actions of the users working with the same drawing would have to be synchronized. To accomplish that, we would have to define a protocol for updating the drawing. The protocol would basically be a set of transactions that are always completed uninterrupted, e.g. a complete dragging operation. The communication between different pdb processes would probably be implemented using remote CORBA [Siegel96] objects.

Extensions to projective three-space

It is natural to ask if the current system can be generalized to three-space. We believe that such a generalization is possible but would involve a number of difficulties.

- The geometric objects will be more difficult to visualize and to interact with than in the two-dimensional case. For example, how should an infinite plane in P_3 be drawn, so that the user can estimate its slope? How should a quadric (the three-dimensional counterpart of a conic) be drawn so that its shape can be clearly perceived? It will probably be necessary to put texture on the surfaces and to compute shades and occlusions. At the same time, the scene must be updated in real-time when objects are dragged. Furthermore, to manipulate three-dimensional objects and to drag such objects around in three-space using a standard pointing device will be quite difficult and will require a carefully designed user interface.

- However, the main difficulty lies in mastering the increased complexity of the algorithms and the data structures. How many different types of objects and constraints should the system be able to handle? It is always a problem



to set the boundaries of a system, but in the two-dimensional case, it is easier to find a reasonably small set of objects that can be combined freely. For example, pdb supports only points, lines and conics. This set is "closed" in the sense that conics and lines intersect in points. In contrast, two quadrics in three-space intersect in a non-planar curve. Should the three-dimensional version therefore be able to handle arbitrary algebraic space curves? If so, why not general algebraic surfaces as well? Should it be possible to define the intersections and the tangent planes of such surfaces? It is evident that the set of different types of objects types and different types of constraints will be significantly larger in the three-dimensional case. The constructive approach using an explicit dependency graph will probably have to be replaced by a more general, algebraic representation of objects and constraints. More general geometric algorithms have to be developed, possibly using Clifford algebra [Hestenes84].

Needless to say, a three-dimensional version of pdb will be orders of magnitude more difficult to write than the two-dimensional version. On the other hand, the work would be very rewarding. Since three-dimensional geometry is so much harder to picture in the mind's eye, a tool for three-dimensional dynamic geometry would be extremely useful. We also believe that the experience we have gained by implementing the current version of pdb will prove to be invaluable if we develop a three-dimensional version.

Appendix A

TIDE: A Scripting Language Interface to C++ Libraries

Abstract

In this paper we discuss how to generate wrapper code that allows existing C++ class libraries to be accessed from scripting languages such as Tcl and Perl. Previously suggested approaches to this problem are reviewed and compared. We point out some problems related to the C++ object model and the difficulties introduced by advanced C++ constructs such as templates, nested types, type definitions, temporaries, implicit casts, multiple inheritance and overloading. We argue that it is necessary to support all of these features, as they are used frequently in all modern C++ libraries. A new system called TIDE, which integrates C++ and Tcl, is presented and we describe the ideas behind its design. As an example, we show how TIDE can be used for accessing the ISO C++ draft standard library.

A.1 Introduction

Interpreted scripting languages like Tcl [Ousterhout94] and Perl [Wall96] are becoming increasingly popular alternatives to strongly typed and compiled languages such as C and C++. Programmers have reported a significant increase in productivity when using scripting languages, especially for applications with graphical user interfaces. There are probably several reasons for that. Scripting languages are simple and easy to learn. They are interpreted and thus encourage experimentation and incremental development. Scripting languages often operate on a higher level than most traditional languages, which makes things like file management and event processing easy to implement.

Many scripting language interpreters are *embeddable*, which means that they can be integrated into any application program. The standard command repertoire can be extended with new, application-specific commands, implemented by

external functions written in C. For example, when using a word processor, it is often convenient to express complicated editing operations in a built-in macro language. When the word processor is launched, it can instantiate a new scripting language interpreter and add application-specific commands such as basic text editing operations, see Figure A.1.

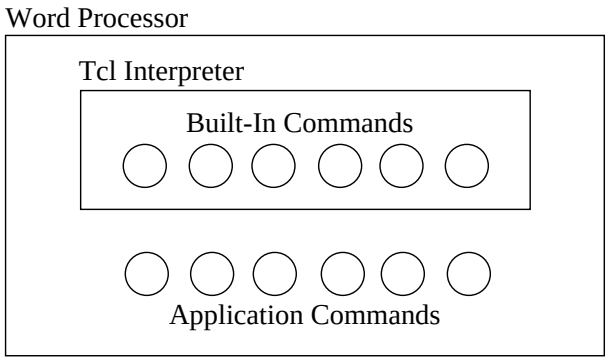


Figure A.1. A word processor with a Tcl interpreter.

The simplest possible application program using an embeddable interpreter is a *shell*. It is a completely general program which simply creates an interpreter, reads commands from an input stream and passes them on to the interpreter for execution. The shell's interpreter will not contain any application-specific commands. Instead, it will be dynamically linked to external application libraries at run-time, and a new command will be added to the interpreter for each library function found. In this way, it is possible to access libraries without ever compiling a main program. For example, if an OpenGL graphics library [Woo97] is dynamically linked to the shell, one can experiment with 3D graphics by writing simple scripts or by typing commands interactively to the shell prompt.

The possibility of extending the core scripting language with new commands implemented as external functions makes an embeddable interpreter a very powerful tool. However, it is usually assumed that extensions will be written in C, and most scripting languages have very little support for other languages. In this paper, we discuss how C++ libraries can be accessed from a scripting language. The use of object-oriented libraries means that classes and objects must be represented in the scripting language and that polymorphism must be supported. Furthermore, we will assume that the C++ libraries have been written *independently* of the scripting language and thus have not been designed to cooperate with an interpreter. This will introduce additional problems that have to do with object identity and memory management, but on the other hand, it will enable us to use any existing C++ library without modifying its source code.

Calling an external function from an interpreter requires a piece of *wrapper code*, written specifically for that function. The interpreter will pass parameters to external functions in a fixed, pre-determined format and it will be the responsibility of the wrapper code to place the parameters in the appropriate registers

or on the stack, before calling the external function. The wrapper code may also have to convert parameters to the representation expected by the external function. For example, the interpreter might pass the integer parameter `17` as a string, `"17"`, while the external function expects to get a binary integer.

Writing such wrapper code by hand is both tedious and error-prone, even for a small library. When classes and polymorphism are involved, the wrapper code gets quite complicated. In this paper, we present a new tool called TIDE (Tcl Interface DEfinition), which can generate the wrapper code automatically. From a declaration of functions and classes in an existing C++ library, which will be called the *client library*, TIDE creates a companion *wrapper code library*. Both the client library and the corresponding wrapper code library can be dynamically linked to the Tcl shell, `tclsh`, or any other application program that contains a Tcl interpreter, see Figure A.2.

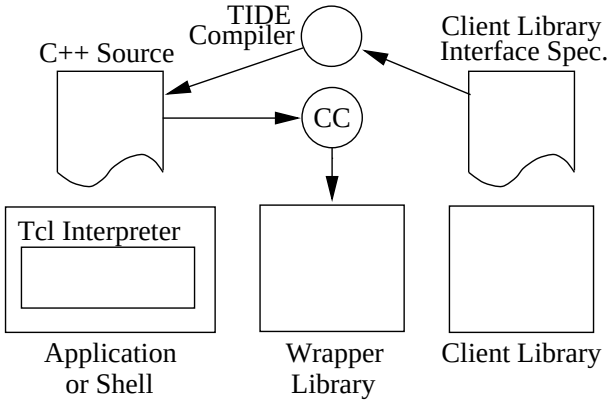


Figure A.2. Compiling a wrapper code library.

Although the focus here will be on Tcl, the same technique could be used with minor modifications for Perl and other scripting languages.

It is assumed that the reader is familiar with C++ but not necessarily with Tcl or Perl. Therefore, a short description of Tcl is given in the next section. Next, we apply TIDE to a simple class and show how the system works in practice. Previous work is reviewed in Section A.4, and in Section A.5, a more complicated class, which cannot be handled by existing tools, is presented. A number of important design decisions are discussed in in Section A.6. Sections A.7, A.8 and A.9 describe how a class interface is specified in TIDE, how the interface can be used from Tcl, and what the wrapper code looks like.

A.2 Tcl – the Tool Command Language

Tcl is an embeddable scripting language and extensions are assumed to be written in C. The language is very simple. All commands have the structure `command arg1 arg2 ...`, where `command` is either a procedure defined in Tcl itself, or

a command implemented by an external C function. The main data types are strings, lists and associative arrays[1]. `xyz` and `17` are both string literals. `xyz` can be used as a variable or procedure name, and `17` can be used as a numerical operand. If `xyz` is the name of a variable, `$xyz` refers to its value while `xyz` refers to the variable itself. For example, `set x $y` assigns the value of `y` to `x`. `{xyz 17}` is a list of length 2, and `{}` is the empty list. Control structures such as `if` and `for` are regular commands which take boolean expressions and command lists as arguments. For instance, the command `while {$x<17} {incr x}` increments the value of `x` until it reaches `17`. If the result of one command is going to be used as an argument to another command, it must be enclosed by `[]`. The C++ statement `g(f(x))` is written `g [f $x]` in Tcl.

To add a new command to a Tcl interpreter, it is sufficient to call a function in the Tcl kernel, specifying the command name and the address of the external function which implements the command. Because string is the fundamental data type in Tcl, the Tcl interpreter passes arguments to external functions as arrays of strings, and expects the functions to return the result as a string. It is usually necessary to add wrapper code which converts argument strings to integers, floating point numbers and other objects.

A.3 Using TIDE – a simple example

Suppose we have a library containing a class `CDPlayer` that allows us to use the Compact Disc unit of a workstation to play music CDs. The class is declared in `cdplayer.hh`, which is shown in Figure A.3.

```
class CDPlayer {
public:
  CDPlayer(string cd, string audio);
  void track(int number);
  void play();
  void stop();
  void eject();
private:
  // ...
};
```

Figure A.3. Declation of class `CDPlayer`.

The constructor takes two arguments which identify the CD driver and the the speaker driver. To access this class from Tcl, we specify its interface in a separate file, `cdplayer.tide`, from which TIDE will generate the wrapper code, see Figure A.4.

[1] Here we describe Tcl 7.6. A richer set of data types will probably be available in Tcl 8.

```
class CDPlayer {} {
    constructor {string string}
    function void track {int}
    function void play {}
    function void stop {}
    function void eject {}
}
```

Figure A.4. TIDE interface to class `CDPlayer`.

The contents of `cdplayer.tide` is actually a small a Tcl program. What each statement means exactly will be explained in detail in Section A.7, but it should be apparent that the file contains the same information as `cdplayer.hh` in this case, and that, in fact, it could be generated automatically. However, in more complicated cases, the `*.tide` files contain information that cannot be derived from the corresponding C++ declarations alone.

The TIDE compiler, `tidec`, is invoked twice. First, the wrapper code for the `CDPlayer` class is generated. Then, a Tcl package which can be loaded dynamically by the Tcl shell is created:

```
tidec cdplayer.tide
tidec -package multimedia cdplayer.tide
```

`tidec` generates C++ source files that are compiled into a shared wrapper code library. The Tcl script in Figure A.5 shows how the CD player can be used. After loading the TIDE library and the wrapper code library, an instance of `CDPlayer` is created and ordered to play a track.

```
load {} Tide
load {} Multimedia
set cd [CDPlayer::CDPlayer& /dev/cd /dev/audio]
CDPlayer::track $cd 1
CDPlayer::play $cd
```

Figure A.5. Using class `CDPlayer` *from Tcl.*

As shown, TIDE commands are written fully qualified with nested names preceded by their enclosing scopes and `::` operators[2]. The reasons for choosing this syntax instead of, say, `$cd track 1`, will be given in Section A.6.

A.4 Previous work

Several attempts have been made earlier to generate wrapper code automatically. For example, the SWIG system [Beazley96] parses C++ header files and turns

[2] When namespaces appear in Tcl, this syntax may be changed.

them into wrapper code for Tcl or Perl. A similar approach is used in Itcl++ [Heidrish94] and vtk [Martin96]. Perl has a built-in extension interface language XS with a C-like grammar. However, these tools only support a small subset of C++, so in practice, they can only be used for libraries with very simple interfaces or libraries that have been *designed* to work with a particular wrapper code generator.

ObjectTcl [Sheehan95] also generates wrapper code automatically from an interface specification, but assumes that the interpreter has full control over the creation and destruction of objects. That is a severe restriction which will be discussed further in Section A.6. In contrast, SWIG, Itcl++ and vtk allow the C++ library to take the initiative to create and destroy objects.

Other solutions, such as the one used in the ABC system [Manges94], require changes to the C++ library code or partly hand-written wrapper code.

Some tools allow scripts to be called from C++ (for example Hush [Eliens95]) or make it possible to subclass existing C++ classes in the scripting language [Sheehan95]. While that certainly can be useful, it is quite different from the problem discussed in this paper.

A.5 A more challenging problem

Let us take a look at the interface of the standard C++ list class [NCITS98]. It is a good example of what kind of code one can expect to find in real-world libraries. Actually, it is very likely that this and other classes from the standard library will be used in the majority of library interfaces in the future, so we simply have to be able to deal with it. Figure A.6 shows the declaration of the `list` class[3]. This interface uses many advanced C++ constructs. For example, the `list` class is a template, parameterized by the element type `T` and a memory management help class, called `Allocator`. To avoid name conflicts, the class has been declared in the standard library namespace, `std`. The class specification contains nested classes (the iterators) and nested type definitions (`typedef`). These nested names occur in other declarations, e.g., in the declaration of `iterator`'s base class `bidirectional_iterator`. Overloading, `const` qualification and reference types (types ending with a `&`) are also used in the specification.

All parts of this interface should be accessible from scripts. If we want to call library functions that take `list` arguments or return `list` values, we have to be able to create such objects and to access them from scripts. None of the wrapper code generators we have tested have been able to cope directly with a class like this one. TIDE does not support every C++ feature either, but it does handle the full `list` class interface, including the nested iterator (generalized pointer) types. We will return to the `list` class and discuss the TIDE interface specification (Section A.7), show how the class can be used in Tcl scripts (Section A.8) and

[3] The declaration has been shortened. Also, the iterator types do not have to be classes according to the draft standard document.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class list {
  public:
    typedef Allocator allocator_type;
    typename allocator_type::size_type;
    typename allocator_type::difference_type;
    typename allocator_type::reference;
    typedef allocator_type::size_type size_type;
    typedef allocator_type::difference_type difference_type;
    typedef allocator_type::reference reference;
    typedef T value_type;

    class iterator :
      public bidirectional_iterator<T, difference_type> {
    public:
      iterator();
      bool operator==(const iterator& x) const;
      reference operator*() const;
      iterator& operator++();
      // ...
    };

    explicit list();
    list(const list& x);
    ~list();
    list& operator=(const list& x);
    iterator begin();
    iterator end();
    size_type size() const;
    iterator insert(iterator position, const T&);
    // ...
  };
}
```

Figure A.6. Declaration of class `list`.

look at the generated wrapper code (Section A.9). But first, we will explain our choice of syntax and object representation in Tcl.

A.6 Representing C++ objects in the interpreter

The scripting language must allow us to create C++ objects, invoke their member functions, and use them as arguments to other C++ functions or Tcl procedures. It must also be possible to store objects or object references returned from C++ functions in Tcl variables. The most important part of the design is the choice of

representation of C++ objects in the Tcl interpreter. In particular, it must be decided whether interpreter resources (variables or procedures) should be allocated to each C++ object. That will have a major impact on both the implementation of the wrapper code and the resulting Tcl interface.

In most object-oriented extensions to Tcl, such as `[incr Tcl]` [McLennan95], objects are represented as Tcl *functions*. An `[incr Tcl]` version of the CD example in Figure A.5 could look like this:

```
CDPlayer cd /dev/cd /dev/audio
cd track 1
cd play
delete object cd
```

Here, `CDPlayer` is a function that represents the CD player class. When it is invoked on the first line, it will create a new function, `cd`, which represents a `CDPlayer` instance. On the second line, `cd` is invoked with the "member function name" `track` as an argument, and `cd` is removed by `delete` on the last line. The syntax feels object-oriented; the user will think of `cd` as an object, not as a function. No other dispatch mechanism than a plain Tcl function call is required. This solution has also been adopted by Itcl++, vtk and ObjectTcl.

Things get more complicated when pointers or references to objects that have not been created by, or previously seen by the interpreter are returned from C++ library functions. In principle, whenever an object is returned from C++ as a function return value or through a parameter, its address must be compared with the address of every C++ object known to the interpreter. If the object has not been seen before, it must be given a Tcl name on the fly. This is more complicated than it may seem, because in the presence of multiple inheritance, a single C++ object can appear to have different addresses depending on the type of pointer used. Therefore, when an object pointer is looked up, it must be converted to every known base class type. This is actually done by some systems, for example Itcl++.

However, this does not solve the whole problem. If a class `D` is derived from two base classes `B1` and `B2`, the address of a `D` instance created by the client library may first be returned as a `B1` pointer (Figure A.7a), then later as a `B2` pointer (Figure A.7b). There is no way the interpreter can see that the two pointers actually refer to the same object, and the interpreter has to create two distinct Tcl names for them. Later, the `D` pointer might be returned, and at that point, the interpreter can tell that all three pointers refer to the same C++ object (Figure A.7c). What should the interpreter do then about the multiple names? This problem is a consequence of the rather weak object model used by C++ [NCITS98].

All of this can of course be avoided if we only allow the Tcl interpreter to create objects. The interpreter will in that case know exactly which class each object belongs to. However, such a constraint is in general not acceptable. Existing C++ libraries often create complicated data structures in a single function call, and various access functions will return references to individual objects in

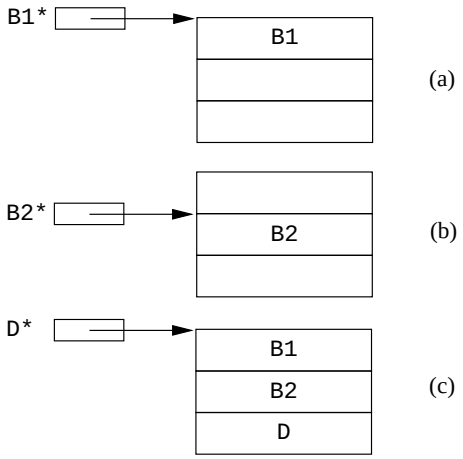


Figure A.7. Unequal pointers may refer to the same (complete) object in C++.

those structures. In fact, it is very common that library classes have no accessible constructors, and only allow so called *factory methods* [Gamma95] to create objects.

An even more delicate problem is to *release* the resources (a Tcl procedure, a hash table entry etc) associated with a C++ object when it is destroyed. Failure to do so will cause the interpreter to grow indefinitely and eventually consume all available memory. The problem is that in general, the interpreter will not know that an object has disappeared unless the object has been destroyed by the interpreter itself. For example, if the Tcl program calls a client library function to destroy a large C++ tree structure, the tree nodes may or may not be destroyed recursively, depending on the semantics of the C++ function. Some of the objects destroyed might also contain members for which resources have been allocated in the Tcl interpreter. Garbage collection on the Tcl side is no option. Since any set of characters can be composed into a Tcl name at any time, it is not possible to verify that a Tcl name is no longer in use just by examining Tcl variables or the Tcl stack.

Because of this, SWIG takes a different approach. In the Tcl interpreter, a pointer to a C++ object is represented by a string containing the object's type and hexadecimal address. For example, a pointer to a `CDPlayer` instance at (hexadecimal) address `aab050` will be represented by the string `_aab050_CDPlayer_p`. The main advantage of using a pure string representation for pointers is that they do not need interpreter resources that later have to be reclaimed.

SWIG references are equivalent to typed C++ pointers, and like C++ pointers, these strings will become "dangling" if the referenced object is destroyed. Just as in C++, the programmer has to take responsibility for not using pointers to objects that no longer exist. Following a dangling pointer will normally result in a core dump.

This approach may seem unappealing, since a mistyped command can cause a

Tcl interpreter to dump core. However, remember that we are actually using C++ libraries that have not been designed to be used from Tcl. The C++ programming model, including the dangling pointer problem, will shine through the thin layer of Tcl bindings that SWIG provides. Even without SWIG pointers, we would still cause a core dump if we called a client library routine and broke its precondition by giving it bad arguments such as an index out of range. Considering this and the fact that the SWIG approach avoids the problem of releasing interpreter resources, we have chosen the same solution in TIDE.

In TIDE, the C++ object and pointer models are exported unmodified to Tcl. Thus, the Tcl commands generated by TIDE comprise a direct, low-level interface to the C++ code. Of course, it is still possible to wrap the TIDE commands in another layer of Tcl procedures to make them safer (avoiding core dumps) or to embed them in a specific object-oriented Tcl extension, such as `[incr Tcl]`. Actually, that would probably be a lot easier to do in Tcl than in C++. In many situations, though, the commands generated by TIDE are safe enough to be used as they are, for example in action procedures invoked through a graphical user interface. In those cases, they will have a minimal run-time overhead.

The decision not to represent C++ objects as Tcl procedures rules out the `<object> <function> <args>` syntax used in `[incr Tcl]` and other systems for calling member functions. Instead, TIDE and SWIG use a straight Tcl procedure call syntax `<function> <object> <args>`. In our opinion, this has the additional advantage of giving member function calls and ordinary function calls identical syntax.

So far we have discussed the string representation of object *references*. It is also possible to represent the *value* (internal state) of some objects[4] as strings. For example, an integer 17 can be represented by the string `17` and the complex number $1 - 3i$ may be written `(1,-3)` or `{1 -3}`. We call these strings *literals*.

It is usually easier to work with literals than object references in Tcl. Built-in Tcl procedures can *only* operate on literals. For example, in the Tcl command `while $condition body`, `$condition` may be `0` or `1`, but not a reference to a boolean C++ object such as `_a3c100_bool_p`. Literals can also be passed to external C++ functions. TIDE will automatically create a temporary C++ object from each literal argument before calling a C++ function. In the fourth command in Figure A.5, `CDPlayer::track $cd 1`, the integer constant `1` will be converted to a temporary `int` object.

However, there are situations when object references must be used. Sometimes the identity of objects matters and some external functions need to return values in reference parameters. There are also many classes that do not have literals, for example streams, structures with cycles, and images.

In TIDE, objects returned from C++ functions are converted to literals by default. However, if an ampersand is appended to a function name, an object reference will be returned instead. The CD player class instantiated in Figure A.5

[4] What is said here about objects and classes also applies to all fundamental C++ types such as characters, integers, and floating-point numbers.

is not likely to have literals, and therefore, the reference-returning version of the constructor, `CDPlayer::CDPlayer&`, was used.

A.7 Specifying the `list` interface

Most of the wrapper code can be generated from the C++ declarations found in header files. In fact, SWIG, Itcl++ and vtk can parse C++ header files. However, none of these packages have implemented the full C++ grammar, so they will fail on complicated declarations. Other systems, such as ObjectTcl, generate wrapper code from interface specification files written in a special language (usually Tcl).

All systems occasionally need more information than the C++ header files can provide. For example, functions might have to be given other names in the interpreter, certain templates have to be instantiated, and more semantic information about parameters or return values might be required. If the wrapper code is generated directly from C++ header files, this information must be given in auxiliary files.

TIDE currently uses the same approach as ObjectTcl; every class and function interface is specified by Tcl commands in a separate file. The specification of the TIDE interface to the standard list class is shown in Figure A.8.

The `use` command says that this specification depends on the TIDE specification of the fundamental types in `tide/fundamental.tide` and on the allocator class template in `tide/defalloc.tide`. The name of the corresponding C++ header file is given by `header`.

Template parameters can be declared for both classes and functions using the `-template` option. The argument is a Tcl list of template parameters, such as `class T`, `const int` or `template<class T> class allocator`[5]. `list` is parameterized by the element type `T` and a help class `Allocator`. The TIDE specification deviates a little from the C++ declaration: here, the class has only one type parameter and `allocator_type` is declared directly as a `typedef` name. This is not at all necessary but simplifies the TIDE specification slightly and is more likely to match the actual C++ declaration used in most existing implementations of the `list` class[6].

Base classes are specified after the class name. `list` itself has no bases, but the nested class `iterator` has. Virtual base classes should be preceded by the keyword `virtual`.

The last argument to `class` is the body which declares all members. The `constructor` command specifies that a constructor with certain parameter types is accessible, and that TIDE may generate wrapper code that depends on it. Similarly, there is a `destructor` command which declares an accessible destructor.

The `function` command declares functions. When used in a `class` body, it

[5] Template template parameters are supported in principle, but have not really been tested since few compilers implement them yet.

[6] Such as the `list` class that comes with GNU `g++` 2.7.2.

```
use tide/fundamental.tide
use tide/defalloc.tide
header list

namespace std {
  class -template {"class T"} list {} {
    typedef allocator<T> allocator_type
    typename allocator_type::size_type
    typename allocator_type::difference_type
    typename allocator_type::reference
    typedef allocator_type::size_type size_type
    typedef allocator_type::difference_type difference_type
    typedef allocator_type::reference reference
    typedef T value_type

    class iterator {bidirectional_iterator<T,difference_type>} {
      constructor {}
      destructor
      function -alias equal bool operator== {"const iterator&"} const
      function -alias index reference operator* {} const
      function -alias incr iterator& operator++ {}
      # ...
    }

    constructor {}
    constructor -alias copy {"const list&"}
    destructor
    function -alias set list& operator= {"const list&"}
    function iterator begin {}
    function iterator end {}
    function size_type size {} const
    function iterator insert {iterator "const T&"}
    # ...
  }
}
```

Figure A.8. TIDE interface to class `list`.

declares member functions. The first argument is the return type, the second is the function name, and the third is a Tcl list of parameter declarations. The keywords `static` and `const` may be used to declare static members (class methods) and constant members, respectively. Functions can be given more descriptive or shorter names in Tcl with the `-alias` option. For example, the C++ operator members `operator=`, `operator*` and `operator++` have been given the "standard" Tcl names `set`, `index` and `incr`, respectively.

Interface specifications can be split over several `*.tide` files, which can be pro-

cessed separately and later combined into a single, dynamically loadable Tcl package. Template instantiation commands are typically given in a separate `*.tide` file:

```
instantiate_class std::list<int>
instantiate_class std::list<std::complex<double>>
```

A.8 Interacting with the interpreter

A.8.1 Using the `list` class

Suppose the Tcl interface of the standard `list` class defined in the previous section has been successfully compiled and loaded into a Tcl interpreter. How can it be used in scripts? Let us say that we have also created a Tcl interface to a C++ graphics library, and that the library contains a matrix class and an abstract base class `Shape` from which all other graphical objects have been derived. Also, assume that the interface of `Shape` contains a virtual (polymorphic) function, `transform`, which applies a projective transformation defined by a 3x3 matrix to a `Shape` object. The Tcl code shown in Figure A.9 creates a transformation matrix and applies it to each element of a vector `v` of `Shape` pointers.

```
set mat [Matrix::Matrix 0.7 -0.7 0 0.7 0.7 0 0 0 1]
for {set i 0} {$i<[std::vector<Shape*>::size $v]} {incr i} {
  Shape::transform [std::vector<Shape*>::index $v $i] $mat
}
```

Figure A.9. Transforming `Shape*` *vector elements.*

The `size` function returns an integer literal, so the loop condition can be expressed with the built-in Tcl operator `<`. It is assumed that the `index` member is an alias for `vector<Shape*>::operator[]`[7].

For simplicity, we used a `vector` instead of a `list` in Figure A.9. Writing the same code for a C++ list is a bit more complicated because list objects have no indexing operator and must be accessed through iterators. The list class has two member functions, `begin` and `end`, which return iterators to the beginning and the end of a list. Since the iterator type has no literals[8], we use the variants `begin&` and `end&` which return object references, see Figure A.10.

The body of the loop in Figure A.10 is repeated until `$iter` points to the end of the list. The `std::list<Shape*>::iterator::index` command (cf Figure A.8) returns the element which `$iter` points to, in this case a `Shape` pointer.

[7] The name `operator[]` is not only clumsy, but the square brackets would have to be escaped in Tcl.

[8] An iterator represents a position in a particular container, which is difficult to represent as a literal value.

```
set mat [Matrix::Matrix 0.7 -0.7 0 0.7 0.7 0 0 0 1]
set iter [std::list<Shape*>::begin& $l]
set end [std::list<Shape*>::end& $l]
while {![std::list<Shape*>::iterator::equal $iter $end]} {
  Shape::transform [std::list<Shape*>::iterator::index $iter] $mat
  std::list<Shape*>::iterator::incr& $iter
}
```

Figure A.10. Transforming `Shape*` *list elements.*

The `Matrix` constructor called on the first line have no trailing ampersand. That means a literal matrix is stored in `$mat`. When that literal is passed to `transform`, a temporary C++ object will be created; using matrix references would be more efficient here. Note that the `equal` call in the loop condition *must* return a literal since the value is tested by the built-in `while` command. Also note that the loop condition must be written in terms of the iterator member function `equal`; a simple string comparison (== in Tcl) will not work.

If possible, TIDE uses the standard C++ I/O functions `operator«` and `operator»` to convert between literals and C++ objects. Should these operators be undefined or unsuitable for a particular class, they may be overridden by C++ conversion functions written by the user. For the list class itself, `list<T>`, it is easy to define literal values provided that the element type `T` has literals. It is natural to choose the same syntax as for built-in Tcl lists, for example `{1 2 3}` for a `list<int>`[9]. The literal conversion functions are easy to write since most of the code is already available as subroutines in the Tcl library. If `$l` is a `list<Shape*>` literal, we can shorten the code in Figure A.10 to

```
set mat [Matrix::Matrix ...]
foreach p $l {
  Shape::transform $p $mat
}
```

A.8.2 Temporaries

Temporaries are created automatically by TIDE when literals are passed as function arguments and when an actual argument needs to be type-cast to match the declaration of a formal parameter. TIDE also creates temporary copies of all objects returned *by-value* from C++ functions. These temporaries live until the `tide_block` in which they were created exits. If there is no surrounding `tide_block`, the temporaries will live until the program exits. Objects created by `new` is guaranteed to live until they are explicitly destroyed by `delete`.

[9] This does not mean that Tcl programs should use the standard C++ list class instead of the built-in Tcl lists. For most Tcl programming, the native Tcl types is a better choice. We are merely demonstrating that if `list` instances are used in the interface of a C++ library, we are able to create and manipulate them.

In the example shown in Figure A.11, the C++ constructor `complex(double real, double imag)` creates a temporary `complex<double>`. The ampersand in the command indicates that a reference to this object is stored in `temp`. On the next line, the copy constructor `complex(const complex&)` makes a copy of `$temp` on the heap. A *pointer* literal referring to the new, dynamically allocated object is returned and stored in `c`. The temporary will be destroyed when `tide_block` exits, while the dynamically allocated copy will live until it is explicitly destroyed by `delete` on the last line.

```
tide_block {
  set temp [std::complex<double>::complex& 1 -3]
  set c [new std::complex<double>::complex $temp]
} ;# $temp is removed here
delete $c
```

Figure A.11. Destruction of temporary objects.

A.8.3 Overloading resolution

The two constructors for the `complex` class in Figure A.11 have the same name; the name is *overloaded*. That is accepted by TIDE as long as the number of arguments differ. If two functions take the same number of arguments, one of them must given a different Tcl name. Type based overloading was considered but rejected for two reasons. First, it would have to be a run-time mechanism with a significant execution time overhead. (In C++ overloading resolution is done at compile time.) Second, literals do not have an inherent type here. In C++, `17` is an `int`, `17.0L` is a `double`, and `"foo"` is a `const char*`. In TIDE, users can define the literal representation for all classes, such as `(1,-3)` for complex numbers and `{1 2 3}` for both `vector<int>` and `list<int>`.

A.8.4 Implicit type conversions

To make the generated Tcl procedures convenient to work with, a number of implicit type conversions have been implemented. If `B` is an unambiguous base class of `D`, a `D` pointer or reference will be implicitly converted to a pointer or reference of type `B`. This also works with multiple inheritance and virtual base classes with the same rules as in C++, and ambiguous implicit casts can be resolved with explicit static casts. Checked down-casts using the C++ `dynamic_cast<>` operator are not directly supported, because TIDE would need information about which base classes of `T` are polymorphic, i.e., has run-time type information, in order to determine whether `dynamic_cast<T*>(x)` would compile. Instead, downcast are

supported by means of a global template function:

```
template<class Base, class Derived>
Derived downcast(Base x) {
  return dynamic_cast<Derived>(x);
}
```

with the TIDE interface

```
function -template {"class Base" "class Derived"} \
  Derived downcast {Base}
```

which can be instantiated by the user for suitable types, e.g., a class `Foo` and its base class `Bar`:

```
instantiate_function Foo* downcast<Bar*,Foo*> {Bar*}
instantiate_function "const Foo*" \
  "downcast<const Foo*,const Bar*>" {"const Bar*"}
```

In Tcl, the resulting commands can then be used in the following way:

```
# instantiate Foo, cast to Bar*
set p [Bar* [new Foo::Foo]]
set q [downcast<Bar*,Foo*> $p]
```

There are also implicit `const` qualification casts, for example from `int*` to `const int*` and from `int**` to `const int* const*`. Any pointer to object can also be implicitly converted to `void*`.

A.8.5 Accessing the TIDE kernel

Since TIDE can be applied to itself, it is possible to create a Tcl interface to the TIDE kernel itself. In fact, Tcl commands for pointer arithmetic, an address operator `&` and a pointer dereference operator `*` have been implemented as Tcl scripts using TIDE kernel functions.

The following code creates a temporary vector `v`. Its address is taken and stored as a literal pointer in `p`.

```
set v [std::vector<int>::vector&]
set p [& $v] ;# the address of $v
```

In a function call, a pointer to an object will be implicitly dereferenced if necessary. (This is a deviation from C++, but a very convenient one.) In the following command, `$p` will be implicitly converted from a `vector<int>` pointer to a `vector<int>` reference:

```
std::vector<int>::resize $p 10
```

A.9 Generating wrapper code

How do one generate wrapper code for a complicated C++ library without ending up implementing a complete C++ compiler? And how can one be convinced that the generated wrapper code is portable and does not rely on a particular object layout or pointer representation?

The approach taken in TIDE is to create a "meta library" with the same structure as the client library we want to access. For every type used in the client library, there will be a corresponding class, called a *meta type*, to represent it. Similarly, every client class will be represented by an instance of a *meta class*. If the client class is a template, so will the meta class be, with a meta type parameter for every type parameter of the client class. In the same manner will every namespace, function and inheritance relationship be represented by meta objects. The idea is to create a meta library that is more or less isomorphic to the client library and let the compiler do all the hard work.

Almost all of the meta objects that together implement the Tcl interface are instances of predefined-defined ordinary classes or template classes. However, each *class* in the client library will be represented by an instance of a unique meta class generated by the TIDE compiler, `tidec`. The meta class will contain very little code, primarily a constructor that instantiates representatives for nested classes and member functions.

The most important meta entity is the meta *type*. Actually, a meta class is a special kind of meta type which represents C++ class types and fundamental types. Meta types for pointers, references and `const` qualified types are created from meta classes and the meta type modifiers `Pointer`, `Reference` and `Const`. For example, the meta class of `int` is `MetaType_int`, and the meta type of `int const*`[10] is `Pointer<Const<MetaType_int»`. The meta type makes the C++ type information available to the TIDE kernel at run-time. It is also responsible for converting between literals and C++ objects, an idea borrowed from the ABC system [Manges94].

The wrapper code[11] generated by the TIDE compiler from `tide/list.tide` (Figure A.8) is shown in Figure A.12. A meta class `MetaType_list` which represents the client class `list` has been generated. To avoid name conflicts, the declaration is nested in the meta counterpart of the `std` namespace. The meta class specification consists of a number of nested `typedef`s, a nested meta class for the iterator, and a constructor. Since `list` takes a type parameter, `MetaType_list` takes a meta type parameter `MetaPar_T`. Template meta classes must take *meta* type parameters because they need the extra information that meta types carry.

Each `typedef` in the client class is mapped to a corresponding meta type definition. If possible, TIDE expands the types. Since `allocator_type` was defined as `allocator<T>` and TIDE had access to the specification of that class template, all nested type names could in this case be reduced to known meta

[10] A pointer to a constant integer.
[11] The code has been shortened.

```
namespace MetaNamespace_std {
  template<class MetaPar_T>
  class MetaType_list :
    public MetaClass<list<MetaPar_T::Type> > {
  public:
    typedef MetaType_allocator<MetaPar_T> MetaType_allocator_type;
    typedef MetaType_unsigned MetaType_size_type;
    typedef MetaType_int MetaType_difference_type;
    typedef Reference<MetaPar_T> MetaType_reference;
    typedef MetaPar_T MetaType_value_type;

    class MetaType_iterator :
      public MetaClass<list<MetaPar_T::Type>::iterator> {
    public:
      MetaType_iterator(MetaScope* scope, const string& name) :
        MetaClass<list<MetaPar_T::Type>::iterator>(scope, name)
      {
        new Inheritance<MetaType_iterator,
          MetaClass<bidirectional_iterator<
            MetaPar_T::Type, int> > >(this, false);
        // ...
        new ConstMemberFunction1<
          MetaType_list<MetaPar_T>::MetaType_iterator,
          MetaType_bool,
          Reference<Const<
            MetaType_list<MetaPar_T>::MetaType_iterator> > >
            (this, "equal",
              list<MetaPar_T::Type>::iterator::operator==);
        // ...
      }
      // ...
    };
```

Figure A.12. Wrapper code for class `list` *(continued on the next page).*

classes and the meta type parameter `MetaPar_T`.

Meta classes are always derived from a TIDE help class, `MetaClass`. The inheritance relationship between *client* classes is represented by the `Inheritance` template class, which is instantiated in the meta class constructors, for example in `MetaType_iterator`. The `Inheritance` constructor takes the meta class object of the derived client class as an argument. The second boolean argument (`false`) says that the client base class `bidirectional_iterator` is non-virtual.

The meta class constructor also instantiates representatives for member functions, e.g., `ConstMemberFunction1` in the `MetaType_iterator` constructor and `MemberFunction1` in `MetaType_list`. There is a separate function class template for each combination of function type (void or non-void, member or non-member)

```
    MetaType_list(MetaScope* scope, const string& unqualified_name) :
      MetaClass<list<MetaPar_T::Type> >(scope, unqualified_name)
    {
      // ...
      new MemberFunction1<
        MetaType_list<MetaPar_T>,
        Reference<MetaType_list<MetaPar_T> >,
        Reference<Const<MetaType_list<MetaPar_T> > > >
          (this, "set", list<MetaPar_T::Type>::operator=);

      new MetaType_allocator<MetaPar_T>(this, "allocator_type");
      new MetaType_unsigned(this, "size_type");
      // ...
    }
    // ...
  };
}
```

Figure A.12. Continued.

and number of parameters. The meta types of the surrounding class, return value and function parameters are passed as template arguments to the function class template.

For each class name alias introduced by a `typedef`, a new instance of the corresponding meta class is instantiated. For example, the name of the nested type `size_type`, which is really an `unsigned` here, is added to the interpreter by the last statement `new MetaType_unsigned(this, "size_type")` in Figure A.12. Thus, the names `unsigned` and `list<T>::size_type` will be synonyms in the Tcl interpreter for every type `T`.

Note that with this scheme, the nested iterator meta class and all member function classes will automatically be specialized and instantiated in the list meta class constructor whenever the list meta class is specialized and instantiated for a particular element type.

References and pointers to objects are represented internally by generic pointers (`void*`). However, all object pointer casts are exposed to the compiler, so that the pointer offset will be adjusted if necessary.

All meta objects are held together by a `Context` instance, which also contains a Tcl interpreter. It is possible to execute Tcl code in different `Context` instances and thus in different interpreters.

A.10 Conclusions and future work

Currently, every TIDE interface specification file has to be written by hand. A C++ parser that can at least partially generate these files will be added in a future

version of TIDE. Furthermore, there are still a number of C++ features not yet supported by the TIDE compiler. For example, TIDE should be able to generate access functions for global variables and member variables. Arrays, pointer to functions and pointer to members should be better supported. (Currently, they have to be wrapped in ordinary classes or `typedef` definitions.) Explicitly specialized templates are accepted, but could be handled more elegantly. However, none of this should be very difficult to implement.

Although TIDE was designed to work with C++ libraries that had been written independently of Tcl, there are situations when one would like such libraries to call a procedure written in Tcl, for example, when implementing *callbacks* or *observers* [Gamma95]. The *library* will then take the initiative to call a function, and the arguments must be converted *from* the C++ representation *to* Tcl strings. The TIDE interface is actually bidirectional, but it will require some additions to the TIDE compiler.

TIDE takes a radically different approach to the wrapper code problem than other systems. Its template based design has proven to be reliable, easy to debug and very flexible. While there are certainly problems to fix and more features to implement, TIDE can handle modern, real-world C++ libraries without requiring source code modifications.

TIDE is available at

```
http://www.nada.kth.se/cvap/tide.html
```

and is distributed under the same terms as the Tcl/Tk source code. The code have been tested with GNU g++ 2.7.2 and Tcl 7.6 under Sun Solaris 2.5.

Bibliography

[Allen93] R. Allen, J. Idt, L. Trilling. Constraint Based Automatic Construction and Manipulation of Geometric Figures. In *Proc. 13th IJCAI (Int. Joint Conf. on Artificial Intelligence)*, Chambery, France, 1993.

[Allen97] R. Allen, L. Trilling. Dynamic Geometry and Declarative Geometric Programming. *Geometry Turned On.* Mathematical Association of America, 1997.

[NCITS98] International Standard ISO/IEC 14882, *Programming Language C++*. National Committee for Information Technology Standards (NCITS), 1998. The document is also available from the American National Standards Institute (ANSI).

[Appelgren94] J. Appelgren. *Reflections – Computer Simulation of the Evolution of Wave Fronts.* Technical Report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm. TRITA-NA-E94/43.

[Arnold98] K. Arnold, J. Gosling. *The Java Programming Language.* Addison-Wesley, 1998.

[Backus97] B. Backus. The Use of Dynamic Software in Teaching and Research in Optometry and Vision Science. *Geometry Turned On.* Mathematical Association of America, 1997.

[Baulac92] Y. Baulac, F. Bellemain, J.M. Laborde. *Cabri – the Interactive Geometry Notebook.* Brooks/Cole Publishing Company, 1992.

[Beazley96] D. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proc. 4th Annual Tcl/Tk Workshop*, USENIX Association, 1996.

[Bix98] R. Bix. *Conics and Cubix.* Springer-Verlag, 1998.

[Bonola55] R. Bonola. *Non-Euclidean Geometry.* Dover, 1955.

[Booch99] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Bouhineau95] D. Bouhineau. Vers une Approche Déclarative pour les Logiciels de Dessins Géométriques. *4èmes journées EIAO (Environnements Interactifs d'Apprentissage avec Ordinateur)*, Cachan, 1995.

[Bouhineau96] D. Bouhineau, S. Channac. La programmation logique par contraintes pour l'aide à l'enseignant. In *Proc. 3rd Int. Conf. on Intelligent Touring Systems*, Montréal, Canada, 1996.

[Channac96] S. Channac. Techniques d'Intelligence Artificielle pour l'Exécution de Programmes Logiques Géométriques. *Journées Infographie Intéractive et Intelligence Artificielle*, Limoges, 1996.

[Cox92] D. Cox, D. O'Shea. *Ideals, Varities and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, 1992.

[Coxeter93] H. Coxeter. *The Real Projective Plane*, 3rd Edition. Springer-Verlag, 1993.

[Coxeter98] H. Coxeter. *Non-Euclidean Geometry*. The Mathematical Association of America, 1998.

[Dahlquist74] G. Dahlquist, Å. Björk. *Numerical Methods*. Prentice Hall, 1974.

[Eliens95] A. Eliëns. *Principles of Object-Oriented Software Development*. Addison-Wesley, 1995.

[Fishback69] W. Fishback. *Projective and Euclidean Geometry*. Wiley, 1969.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Geometry-Center98] Geometry Center at the University of Minnesota. `http://www.geom.umn.edu/java`. December, 1998.

[Goldberg83] A. Goldberg, D. Robson. *Smalltalk-80*. Addison-Wesley, 1983.

[Hamson97] T. Hamson. Beginning Geometry at Collage. *Geometry Turned On*. Mathematical Association of America, 1997.

[Heidrish94] W. Heidrish, P. Slusallek, H. Seidel. Using C++ Class Libraries from an Interpreted Language. In *Proc. TOOLS USA*, 1994.

[Hestenes84] D. Hestenes, G. Sobczyk. *Clifford Algebra to Geometric Calculus*. Reidel, 1984.

[Heydon94] A. Heydon, G. Nelson. *The Juno-2 Contraint-Based Drawing Editor*. SRC Research Report 131a, Digital Equipment, 1994.

[Hägglöf95] K. Hägglöf, P.O. Lindberg, L. Svensson. Computing Global Minima to Polynomial Optimization Problems Using Gröbner Bases. *Global Opt.* 7, 1995.

[Jackiw95] N. Jackiw. *The Geometer's Sketchpad version 3.0.* Key Curriculum Press, 1995.

[King97] J. King. Preface of *Geometry Turned On.* Mathematical Association of America, 1997.

[Klein25] F. Klein. *Elementarmathematik vom höheren Standpunkte aus*, Bd 2. Springer, 1925.

[Klein28] F. Klein. *Vorlesungen über nicht-euklidische Geometrie.* Berlin, 1928.

[Kutzler86] B. Kutzler, S. Stifter. On the Application of Buchberger's Algorithm to Automated Geometry Theorem Proving. *Journal of Symbolic Computing*, Vol. 2, 1986.

[Leon86] S. Leon. *Linear Algebra with Applications.* Macmillan, 1986.

[Manges94] J. Manges, B. Ladd. Tcl/C++ Binding Made Easy. In *Proc. 2nd Annual Tcl/Tk Workshop*, USENIX Association, 1994.

[Manocha94] D. Manocha. Solving Systems of Polynomial Equations. *IEEE Computer Graphics and Applicationns*, March, 1994.

[Martin96] K. Martin. Automated Wrapping of a C++ Class Library into Tcl. In *Proc. 4th Annual Tcl/Tk Workshop*, USENIX Association, 1996.

[McLennan95] M. McLennan. The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More. In *Proc. 3rd Annual Tcl/Tk Workshop*, USENIX Association, 1995.

[Meserve59] B. Meserve. *Fundamental Concepts of Geometry.* Addison-Wesley, 1959.

[Monagan98] M. Monagan, K. Geddes, K. Heal, G. Labaln, S. Vorkoetter. *Maple V Programming Guide.* Springer-Verlag, 1998.

[Naeve86] A. Naeve, J.O. Eklundh. On Projective Geometry and the Recovery of 3-D Structure. In *Proc. 1st Int. Conf. on Computer Vision*, London, England, 1987.

[Naeve89] A. Naeve. *Geometric Modelling – a Projective Approach.* Technical Report CVAP63, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm. TRITA-NA-P8918.

[Naeve93] A. Naeve. *Focal Shape Geometry of Surfaces in Euclidean Space.* Ph.D. Dissertation. Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm. TRITA-NA-P9319.

[Ousterhout94] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Quercia93] V. Quercia, T. O'Reilly. *X Window System User's Guide*. O'Reilly, 1993.

[Requicha80] A. Requicha. Representations for Rigid Solids: Theory, Methods and Systems. *ACM Computing Surveys*, 12(4), 1980, pp 437–464.

[Richter-Gebert95] J. Richter-Gebert. Mechanical Theorem Proving in Projective Geometry. *Annals of Mathematics and Artificial Intelligence*, 13, 1995, pp 139–172.

[Richter-Gebert98] J. Richter-Gebert, U. Kortenkamp. Downloadable demo version of the Cinderella Café software. `http://www.cinderella.de/Demo/Standalone/cindy.html`. November, 1998.

[Richter-Gebert99] J. Richter-Gebert, U. Kortenkamp. *The Interactive Geometry Software Cinderella* (CD-rom with booklet). Springer-Verlag, 1999.

[Salmon60] G. Salmon. *Treatise on Conic Sections*, 6th Edition. Chelsea Publishing Company, 1960.

[Samuel88] P. Samuel. *Projective Geometry*. Springer-Verlag, 1988.

[Schroeder96] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit*. Prentice Hall, 1996.

[Schumann94] H. Schumann, D. Green. *Discovering Geometry with a Computer*. Studentlitteratur, 1994.

[Sheehan95] D. Sheehan. Interpreted C++, Object Oriented Tcl, What next? In *Proc. 3rd Annual Tcl/Tk Workshop*, USENIX Association, 1995.

[Siegel96] J. Siegel (Editor). *CORBA Fundamentals and Programming*. Wiley, 1996.

[Stolfi91] J. Stolfi. *Oriented Projective Geometry*. Academic Press, 1991.

[Stroustroup97] B. Stroustroup. *The C++ Programming Language*, 3rd Edition. Addison-Wesley, 1997.

[Wall96] L. Wall, T. Christiansen, R. Schwartz. *Programming Perl*, 2nd Edition. O'Reilly, 1996.

[Wilson99] D. Wilson. *An Exploration into the Teaching and Learning of Geometry and the Possible Effects of Dynamic Geometry Software*. Available on-line at `http://s13a.math.aca.mmu.ac.uk/Daves_Articles/PI/Contents.html`, January 1999.

[Winger62] R. Winger. *An Introduction to Projective Geometry*. Dover, 1962.

[Winroth94] H. Winroth. Object-Oriented Communication in Image Processing Systems. In *Experimental Environments for Computer Vision and Image Processing*, World Scientific, 1994.

[Winroth98] H. Winroth. A Scripting Language Interface to C++ Libraries. In *TOOLS 23*, IEEE Computer Society Press, 1998. (Also in the appendix of this thesis.)

[Wolfram96] S. Wolfram. *The Mathematica book*, 3rd Edition. Cambridge University Press, 1996.

[Woo97] M. Woo, J. Neider, T. Davis. *OpenGL Programming Guide*, 2nd Edition. Addison-Wesley, 1997.

[Wu86] W-T. Wu. Basic Principles of Mechanical Theorem Proving in Elementary Geometries. *Journal of Automated Reasoning*, Vol. 2, 1986.