A Display System for the Projective Plane

by

Don Tiessen

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Masters of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1994

ISBN 0-612-16336-9

Name ______________________________

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science | 0 | 9 | 8 | 4 | U·M·I

SUBJECT TERM | SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce-Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

A DISPLAY SYSTEM FOR THE PROJECTIVE PLANE

BY

DON TIESSEN

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Abstract

Projective geometry is often introduced to new students as the study of lines and points in a projective plane and the incidences between them. Diagrams are drawn with these lines and points to illustrate the properties of projective geometry in the projective plane. This work describes a computer application that can aid in the creation of projective drawings. It allows users to arbitrarily place projective lines and points in a window representing the real projective plane, to create incidences between them, and to view the projective plane from different points of view. As well, *any* line or point drawn by the user can be grasped and moved with the mouse while the incidence structure of the diagram is maintained automatically.

Of these operations, the process of moving objects in projective drawings was found to be the most interesting to study. This was because of the dependencies between objects created by the incidence structure of the diagram. Much of this thesis is devoted to the theory and implementation involved in solving this problem.

Contents

Chapter 1

Introduction

This thesis describes the development and implementation of a computer application designed to aid in the study of the projective plane. If Euclidean geometry is thought of as the study of the way things are, then projective geometry can be thought of as the study of the way things appear. For example, we would probably use Euclidean geometry to describe the size, position and shape of a vase sitting on a table. Projective geometry would allow us to determine how the vase and table would appear to a person looking at them. This is because of the significance of the *projective transformation* (see Section 2.2) to projective geometry. This transformation closely resembles the mechanism that transforms the three-dimensional world around us into a two-dimensional image in the human eye.

Projective geometry is primarily concerned with the incidences between projective objects in projective space. If the projective space has two dimensions, i.e. the projective plane, then the objects are projective lines and points. If the space has three dimensions, i.e. projective three-space, then the objects are projective lines, points and planes.

There are two ways to study the projective plane: Synthetically and analytically. Synthetic projective geometry is based on a set of axioms and results that can be proven from those axioms. Analytic projective geometry is based on a numerical model of projective space and the algebraic manipulation that can be performed on this model.

The purpose of this thesis is to create a computer application that works with the projective plane. The application appears to the user as a piece of paper on which he or she can draw projective constructions. The application can place projective lines and points in several ways, some of which enforce the axioms of the projective plane. Thus the application may be said to work with synthetic projective geometry. On the other hand, since it is a computer application, it uses analytic projective geometry to work with the projective plane. Thus it, in fact, works simultaneously in synthetic and analytic projective geometry.

This application is fashioned as a graphical editor, working with projective lines and points in the projective plane. Its purpose is to allow a user to build up a construction of projective lines and points and incidences between lines and points, and to perform several operations on that construction. One of the more interesting and fundamental operations is moving one of the objects in the construction. While at first this may seem straight-forward, when the incidence information is considered we find that moving one object can have far-reaching side effects on the other objects in the construction. Thus, as the user moves one object, the effects of the movement are animated throughout the construction in real-time. The ability to manipulate a drawing in real time has been shown to be beneficial to the understanding of geometric concepts [6]. Much of this thesis is devoted to the solution to this problem.

Before continuing, we now discuss some of the applications that the study of projective geometry can lead to. Suppose we had drawn two straight lines on a piece of paper such that, if they were extended off of the page, they would intersect at a point which is not on the page. Using planar projective geometry (the study of projective geometry in a plane), we could make this point of intersection accessible. For example, we could find a third line on the page that passes through the point of intersection, even though the point itself does not appear on the page. This is an example of the use of synthetic projective geometry, because the method uses the Desargues' theorem, described in Section 2.5.

Now suppose that we are writing a computer application that will display a two-dimensional image of a cube on an output device of the computer (i.e. a CRT screen). First, we give the computer a numerical description of the cube. Perhaps this description is a list of the coordinates (in Euclidean three-space) of the vertices of the cube. Then we apply a projective transformation that takes these coordinates to coordinates on a projective plane which represents the output device. This transformation can be applied by multiplying the coordinates in three-space by a matrix. Since we are working with a numerical model of projective space, this is an example of analytic projective geometry.

In Chapter 2, we give a brief overview of projective geometry, both synthetic and analytic. The purpose of this is two-fold. First, it is important to explain the aspects of projective geometry with which the application will be working. Secondly, we need to introduce some of the analytic techniques used by the application.

In Chapter 3 we will introduce the Determining Set. This is work that was developed to aid in the understanding and the solution of the problem of moving one object in a construction. This chapter includes work that allows us to calculate the size or *dimension* of determining sets. In Chapter 4 we will show how determining sets are used, as well as another technique for moving objects. In Chapter 5 we will discuss a partial solution to the problem of moving constructions which have no determining sets, thus allowing us to modify more types of constructions than we would with determining sets alone.

In Chapter 6 we begin discussing the implementation of the application. First we begin with an overview of the Unidraw class library used in the implementation. This is followed by a brief discussion of object orientation and how it was applied to the implementation. Finally we discuss the data structures and algorithms of the implementation that deal with the projective plane.

In Chapter 7 we give a brief user guide to the application, followed by empirical results gained from using the application in Chapter 8.

Chapter 2

An Overview of Planar Projective Geometry

Euclidean geometry is the study of the properties of geometric objects such as lines, line segments, and points. The properties studied include the lengths of line segments, the angles of intersection between lines, the congruence of line segments, etc. The element that each of these properties has in common is that they are all measures.

In contrast, planar projective geometry is the study of points and lines and the incidences of points and lines. The concept of measure is meaningless[1] here. For example, suppose two lines intersect at a point. In the projective plane, the position of the point or the angle between the lines is irrelevant. It is enough to know that two lines intersect at a point.

As another example, consider the triangle. In Euclidean geometry, we could discuss the lengths of the sides of the triangle, whether it is equilateral or scalene, and if it contains a right angle we can discuss it using trigonometry. In planar projective geometry, a

[1]The length of a segment is not defined in planar projective geometry, but the *cross ratio* of the segments determined by four collinear points is an invariant under projective transformation. No more will be said about cross ratios in this thesis.

triangle is simply three (non-concurrent) lines. We are not concerned with their position, as long as there are three of them and they have three distinct points of intersection.

Planar projective geometry can be encapsulated in the following three axioms [8]:

- Any two distinct points are contained in one and only one line.
- Any two distinct lines have in common one and only one point.
- There exist at least four points, no three of which are contained in one line.

While at first it seems that the study of a geometry such as this (with out any metrics) seems dull and pointless, there is an extra capability that this simplicity buys us. If we draw two parallel lines on a piece of paper, Euclidean geometry states that they do not intersect (or that they intersect at infinity). Planar projective geometry, however, has as an axiom that any two distinct lines intersect at a point. The fact that the point of intersection is off the page is at most an inconvenience. We gain access to points at infinity, and they turn out to be no different from other points. Any point can be considered a point at infinity.

The axioms don't specify the number of points or lines, or the way they are represented. A projective plane may contain an infinite or finite number of points and lines. It could be represented (analytically) as a plane over reals, rationals, complex numbers, etc. This thesis deals with the real projective plane. In the rest of this chapter we provide a brief overview of topics in the projective plane.

2.1 Basic Configurations

With the axioms we can construct planar projective geometry configurations. The simplest configuration is the *triangle*. It is created by placing three non-collinear points on the plane and drawing lines containing each pair of points. See Figure 2.1. Note that it

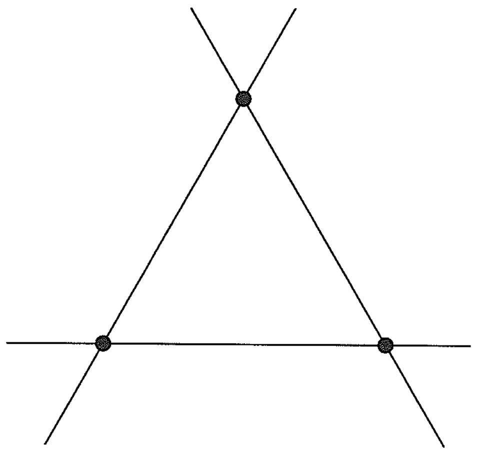

Figure 2.1: A triangle in Planar Projective Geometry.

can be drawn as either three points and the three lines that contain them, or as three lines and the three points of intersection. The latter (dual) form is a *trilateral.*

Another common configuration is the *quadrangle.* This consists of four points on the plane, no three of which are collinear. From these four points we can draw six lines (one through each pair of points). In fact, each pair of points defines two lines; the line defined by the pair, and the line defined by the remaining two points. The two lines thus defined are known as *opposite* lines.

In the above six lines there are three pairs of opposite lines. Each pair of opposite lines intersect in a point, giving three points. These three points are the *diagonal points* of the quadrangle. The triangle formed by the diagonal points is the *diagonal triangle* of the quadrangle. See Figure 2.2.

A dual configuration is the *quadrilateral,* which has four lines, no three of which are concurrent. From these four lines we can draw six points (as the intersection of each pair of lines). In these six points there are three pairs of opposite points. The three lines containing these pairs are the *diagonal lines* of the quadrilateral. The trilateral formed by the diagonal lines is the *diagonal trilateral* of the quadrilateral. See Figure 2.3.

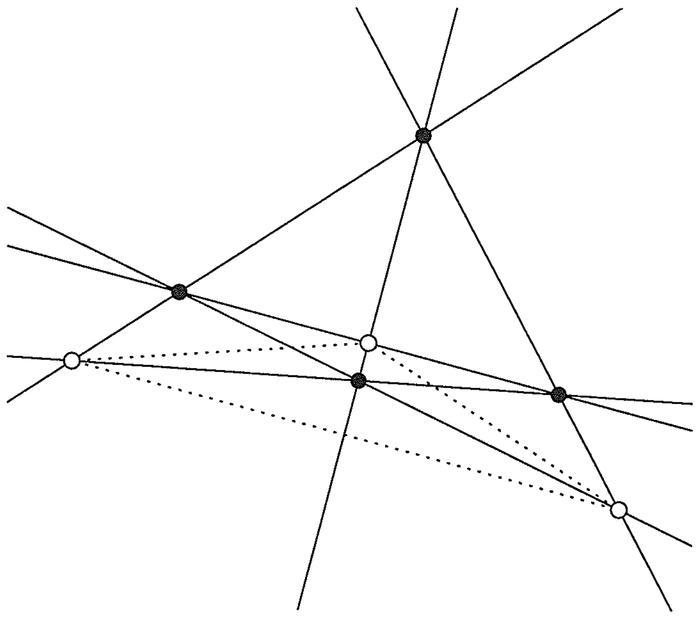

Figure 2.2: A Quadrangle with diagonal points and diagonal triangle.

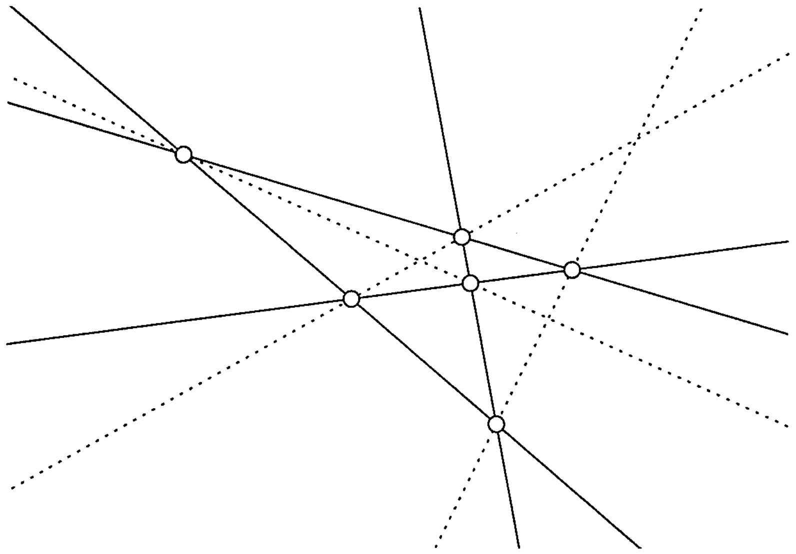

Figure 2.3: A Quadrilateral with diagonal lines and diagonal trilateral.

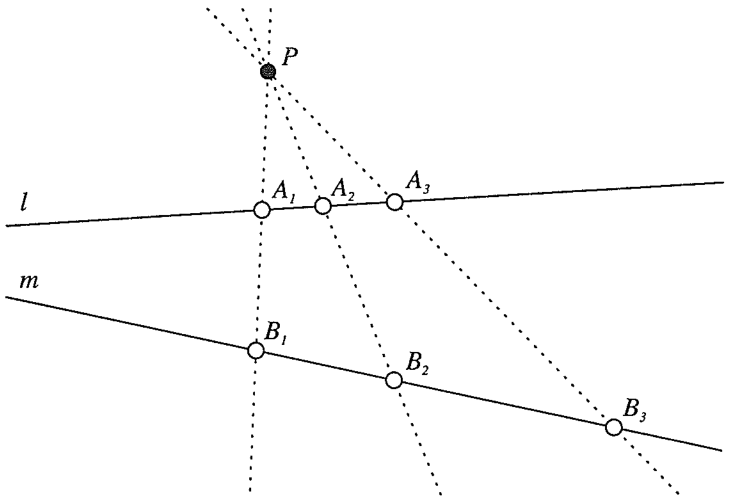

Figure 2.4: A Perspectivity.

2.2 Perspectivities and Projectivities

The fundamental transformation in planar projective geometry is the *perspective* transformation. A perspective transformation takes a set of points on an object line to a set of points on an image line. One parameter for the transformation is a point, known as the *centre of perspectivity* or the *eye*. If a point P is the centre of perspectivity, then each point A on the object line is transformed to a corresponding point A' on the image line, such that the points P, A, and A' are all collinear.

For example, Figure 2.4 shows a perspective transformation with centre P, which takes points A_1, A_2, and A_3 on line l to points B_1, B_2, and B_3, respectively, on line m.

It is also possible to chain perspectivities together to form *projectivities* or *projective* transformations. See Figure 2.5. In this example, A is projected from P onto A', and A' is projected from P' onto A''. This projection is written as $(A)\overline{\wedge}(A'')$ or as $(l)\overline{\wedge}(l'')$.

We now introduce some fundamental results involving projectivities [12]. Any projectivity that takes three distinct, arbitrarily chosen points from one line onto three distinct, arbitrarily chosen points on another line can be performed as a sequence of at most two

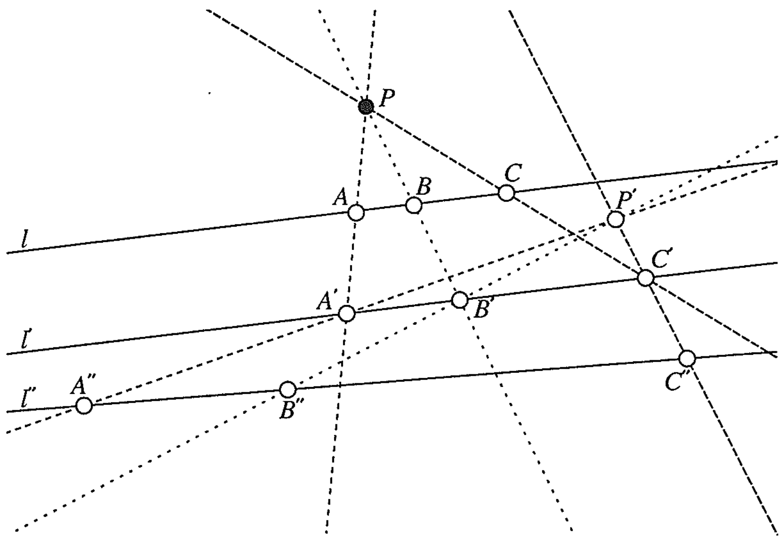

Figure 2.5: A Projectivity.

perspectivities. As well, any projectivity that takes three distinct, arbitrarily chosen collinear points onto three distinct, arbitrarily chosen collinear points can be performed as a sequence of at most three perspectivities. This second case only occurs when the projectivity transforms points on one line to points on the same line. For example, Figure 2.6 shows the projectivity $(A, B, C)\overline{\wedge}(A''', B''', C''')$. The sequence of perspectivities is $(A, B, C) \overset{P}{\overline{\overline{\wedge}}} (A', B', C') \overset{P'}{\overline{\overline{\wedge}}} (A', B'', C''') \overset{P''}{\overline{\overline{\wedge}}} (A''', B''', C''')$. A unique projectivity exists that takes any three given points on one line to any three given points on another line. This statement is known as the Fundamental Theorem of Projective Geometry.

While projectivities are usually considered to transform objects in the projective plane, it is also possible to projectively transform planes in projective three-space. This is discussed in [9].

The *period* of a projectivity is the minimum number of times it must be successively applied in order to achive an identity transformation. An *involution* is a projectivity of period two from a line onto the same line. Thus, applying the projectivity twice in succession returns the points on the line to their original positions.

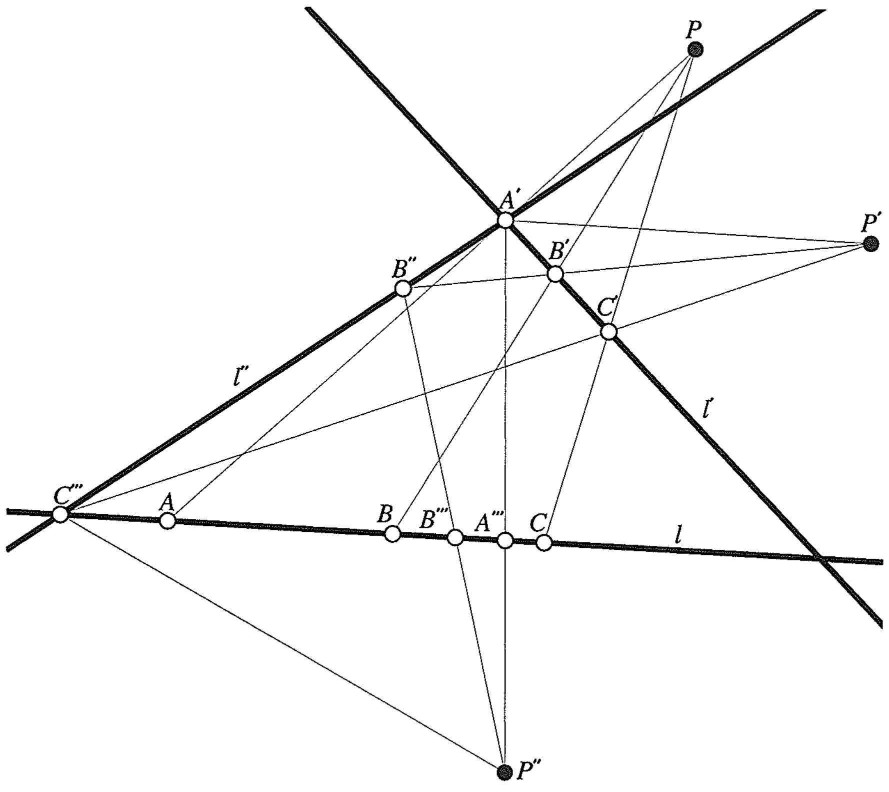

Figure 2.6: A Projectivity onto the same line.

2.3 Duality

In section 2.1, each configuration mentioned had a closely related dual (eg. quadrangle and quadrilateral). These duals are created by simply interchanging points and lines. There is also duality in the axioms of projective geometry given earlier in the chapter: the first two axioms express the incidence of two points and a line, and two lines and a point. This concept of *duality* is fundamental to projective geometry. It further enforces the idea that the incidence structure of a diagram is more important that its appearance.

Essentially, the principle of duality states that for any theorem involving points and lines there exists a dual theorem involving lines and points. The dual theorem is arrived at by simply interchanging the words 'point' and 'line' in the original to 'line' and 'point' respectively in the dual, and then adjusting the other components so that the dual theorem is grammatically correct.

2.4 Harmonic Conjugates

Consider the quadrangle $QPRS$ shown in figure 2.7. The line l passes through two of the diagonal points, A and B. The intersections C and D of the remaining two diagonals with l are known as *harmonic conjugates* with respect to A and B.

In fact, given points A, B and C and the line l, any quadrangle will give the same point D. This is illustrated in Figure 2.8, where each of the four drawings have different quadrangles but have the same four collinear points. This can't be proven from the axioms given. In the real plane it follows from Desargues' theorem, which is described in section 2.5.

When given three collinear points A, B, and C on a line l, it is possible to find a quadrangle that yields the harmonic conjugate D of C with respect to A and B. (See again Figure 2.7). The construction is as follows: select a point P, not on l. Now select

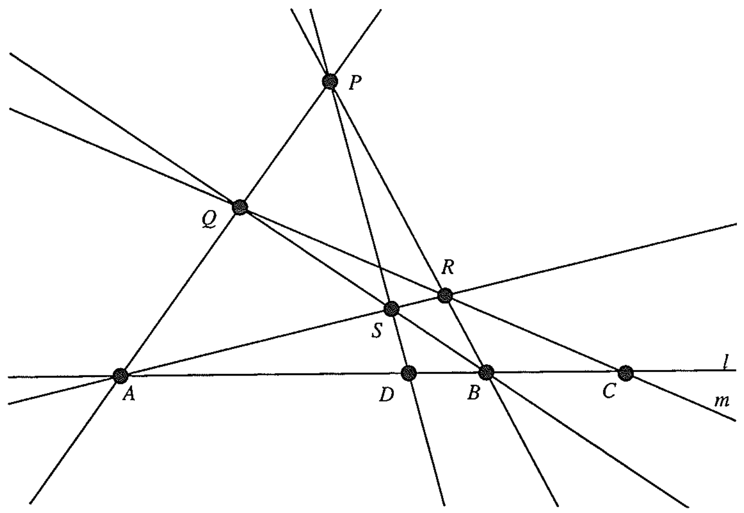

Figure 2.7: Harmonic Conjugates.

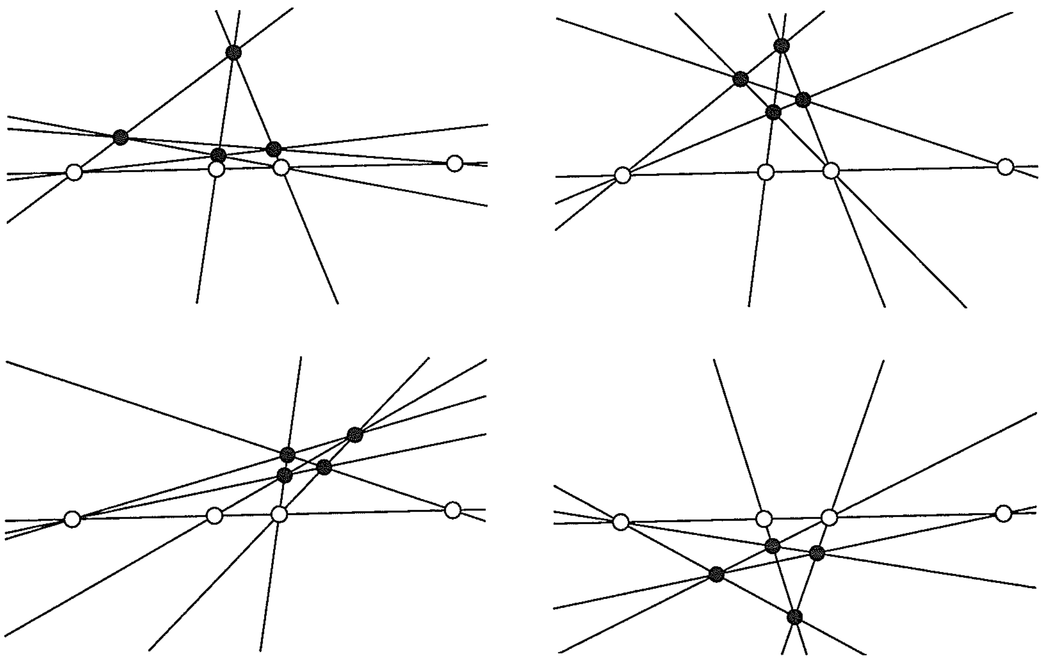

Figure 2.8: Harmonic Conjugates with Varying Quadrangle.

a line through C that is distinct from l and does not contain P, call it m. m intersects AP and BP in points Q and R respectively. The lines AR and BQ intersect at a point S. The point where PS intersects l is D, the harmonic conjugate of C with respect to A and B.

2.5 Desargues' Configuration

Two triangles are said to be in perspective from a point if the lines formed by corresponding vertices of the triangles are concurrent at that point. A fundamental configuration is the *Desargues'* configuration. It is described by Desargues' theorem: *If two triangles are in perspective from a point, then the points determined by the intersections of corresponding sides of the triangles are collinear.* See Figure 2.9. The Desargues' configuration is so fundamental to projective geometry that it is usually taken as an axiom. Even so, I will present a simple proof for Desargues' theorem. The proof requires three dimensions, and so is only valid when three or more dimensions are available. For developing the geometry of the plane (restricted to two dimensions) the proof does not hold, so Desargues' theorem must be taken as an axiom. There exist finite planes, known as *non-Desarguesian planes*, that do not satisfy Desargues' theorem.

2.5.1 A Proof in Three Dimensions.

One simple way to prove Desargues' theorem (given in [8]) works in three dimensions. Suppose we are given two triangles, ABC and $A'B'C'$, which are in perspective from a point V.

In the case where the two triangles are not in the same plane, we can say that the points V, B, B', C, C' are coplanar, since the lines BB' and CC' both pass through V. We can then say that the lines BC and $B'C'$ are also in this plane, and that their point of intersection also lies in planes ABC and $A'B'C'$. Similarly, we can show that the

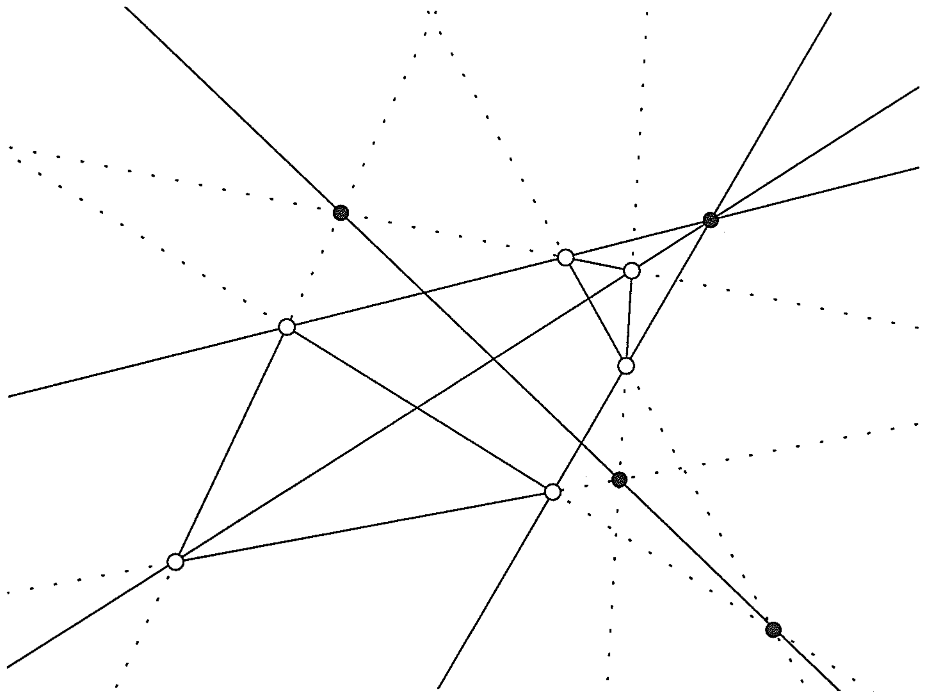

Figure 2.9: The Desargues' Configuration.

intersections of lines AB and $A'B'$, and of lines AC and $A'C'$ also lie in planes ABC and $A'B'C'$. Therefore the three points of intersection are collinear, with their common line being the line of intersection between planes ABC and $A'B'C'$.

The case where both triangles are coplanar is handled by creating a third triangle that is not in the same plane as the other two, but in perspective with both ABC and $A'B'C'$. We then apply the previous case twice to show that Desargues' theorem holds for ABC and $A'B'C'$.

Let π be the plane containing the triangles ABC and $A'B'C'$, and the point of perspection, V. To create the third triangle, we choose a line through V that doesn't lie in π, and two points on this line, S and S', which are distinct from each other and from V.

The lines SA and $S'A'$ are coplanar, and we will call their intersection A^*. Similarly, find points B^* and C^*. Notice that the points A^*, B^*, and C^* are not collinear, and therefore form a triangle. (If A^*, B^*, and C^* were collinear, then A, B, and C would also have to be collinear, and we know this is not the case.) Note that triangles ABC and $A^*B^*C^*$ are not in the same plane, and in perspective with S. Therefore, from the previous case, we know that the intersections of corresponding sides of ABC and $A^*B^*C^*$ will be collinear, and their common line is the line of intersection between the planes π and $A^*B^*C^*$.

Similarly, the triangles $A'B'C'$ and $A^*B^*C^*$ are in perspective from S', so the intersections of their corresponding sides will be collinear, with their common line is the line of intersection between planes π and $A^*B^*C^*$. Since both pairs of triangles have the same line in common, and since corresponding sides of triangles ABC and $A'B'C'$ intersect this line in the same places, we know that the intersections of corresponding sides of triangles ABC and $A'B'C'$ are collinear. Therefore, Desargues' theorem holds in this case as well.

2.5.2 Desargues' from Five Points in Three-Space.

Given five points in general position in three-space (i.e. no three points collinear), it is possible to view them in a way that yields a Desargues' configuration. First, notice that in the five points there are $\binom{5}{2}$ or 10 pairs of points that define 10 distinct lines. Similarly, there are $\binom{5}{3}$ or 10 triples of points that define 10 distinct planes.

Now let π be another plane in three-space that contains none of the five original points. The 10 lines created by the five points will intersect π as 10 distinct points, and the 10 planes created by the five points will intersect π as 10 distinct lines, if π is suitably chosen. Also, in the three points that define each plane, there are $\binom{3}{2}$ or three lines that are coplanar with that plane, which will therefore appear on π as three points on the line given by that plane. Similarly, with each pair of points that defines a line, there are three planes that contain that pair. These planes will appear on π as three lines passing through the point defined by that pair.

The figure now visible on π has 10 lines and 10 points, with three points on every line and three lines through every point, and it is the Desargues' configuration. Therefore the Desargues' configuration can be viewed as the incidence of the $\binom{5}{2}$ pairs on the $\binom{5}{3}$ triples of 5 points. This implies much symmetry in the Desargues' configuration. This construction was suggested by Professor N. Mendelsohn.

2.5.3 A Proof using Desargues'

In section 2.4 we mentioned that, given points A, B, and C on a line l, any quadrilateral would yield the same harmonic conjugate D. This will now be proven with the help of Desargues' Theorem.

Suppose we are given a line l with points A, B, and C. See Figure 2.10. Now suppose we find the harmonic conjugate of C with respect to A and B using two distinct quadrilaterals, $QRST$ and $Q'R'S'T'$. Label the resulting harmonic conjugates D and D',

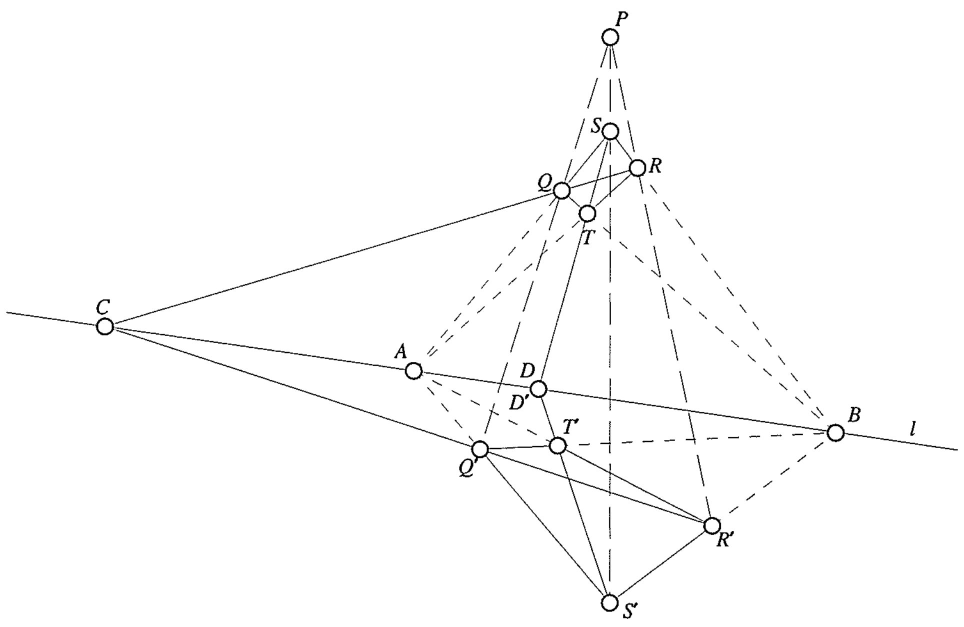

Figure 2.10: A Proof Using the Desargues' Configuration.

respectively. If the quadrilaterals are labeled as in Figure 2.10, we can see that lines QS and $Q'S'$ intersect at A, lines RS and $R'S'$ intersect at B, and lines QR and $Q'R'$ intersect at C. Since corresponding sides of triangles QRS and $Q'R'S'$ intersect in three collinear points, the dual of Desargues' Theorem tells us the triangles are in perspective from a point, say P. That is, lines QQ', SS', and RR' intersect at point P.

Similarly, we can show that the triangles QRT and $Q'R'T'$ are in perspective from the same point P. We now have lines QQ', RR', SS', and TT' all passing through the point P, and can say that the triangles RST and $R'S'T'$ are in perspective from P. Therefore, their corresponding sides meet in three collinear points. Since lines RT and $R'T'$ meet in A, and lines RS and $R'S'$ meet in B, then lines ST and $S'T'$ must meet in a point on l. Since lines ST and $S'T'$ meet l in points D and D', respectively, we can say that D and D' are in fact the same point.

Note that this proof is not complete. In order to be complete, it would have to address all possible ways of situating the second quadrilateral.

2.6 Configurations

Previously the word configuration was used to describe certain planar projective geometry drawings. If in a drawing every line has m points on it, and every point has n lines passing through it, then that drawing is known as a (m, n)-configuration. Thus the Desargues' configuration is a (3,3)-configuration, and a quadrilateral with its diagonal points is a (3,2)-configuration.

Notice that the Desargues' configuration has the same number of lines (10) as it does points. A (m, m)-configuration with K lines and points is referred to as a K_m-configuration. Thus the Desargues' configuration is known as a 10_3-configuration.

2.7 Pappus Configuration

Choose any three points on each of two lines and pair them. Then draw a line through each point on one line and the points on the other line with which it is *not* paired. Of the six resulting lines, the pairs of corresponding lines intersect in three points. These points are collinear. This is Pappus's Theorem. The resulting 9_3-configuration is known as the *Pappus Configuration.* See figure 2.11.

2.8 Incidence Graphs

Since planar projective geometry is primarily concerned with the incidence structure of a set of lines and points, a construction in the projective plane lends itself well to representation as a graph. If we let the nodes of the graph represent projective lines and

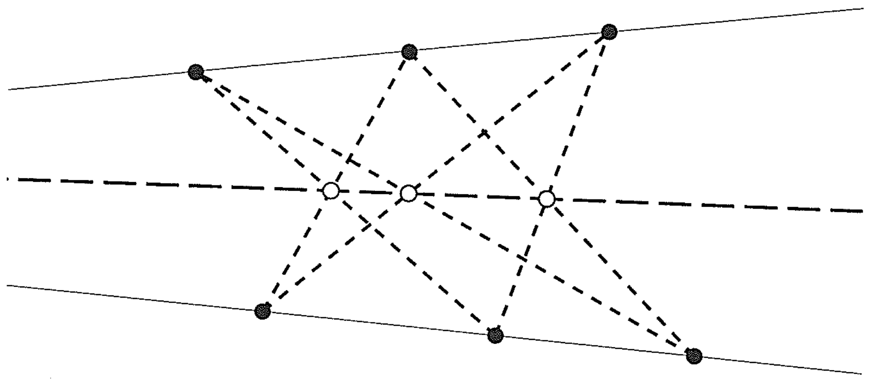

Figure 2.11: The Pappus Configuration.

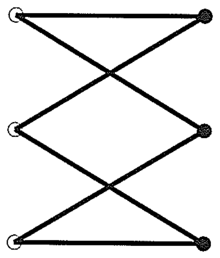

Figure 2.12: A Triangle Represented as a Bipartite Graph.

points, the edges of the graph will be incidences of lines and points. Such a graph created for a given drawing is the *incidence graph* for that drawing.

As well, since we will never have edges between points or between lines, the graph will be a bipartite graph with one bipartition being the set of points and the other bipartition being the set of lines. For example, figure 2.12 shows the bipartite graph corresponding to the triangle shown in figure 2.1.

2.9 Collineations

A *Collineation* is a mapping in the projective plane of points onto points and lines onto lines, such that the incidence structure of the original plane is maintained after the mapping. We will use the notation of [8] where, if a collineation α maps a point P to a point P', we write

$$P \to P' = P^{\alpha}$$

.

If $P = P^{\alpha}$ for a given point P and a given collineation α, P is said to be a *fixed point* under α. Similarly, if $l = l^{\alpha}$ for a given line l and a given collineation α, l is said to be a *fixed line* under α. The intersection of two fixed lines is a fixed point, and the line containing two fixed points is a fixed line. Note that the points on a fixed line are not necessarily fixed. Thus a line of fixed points is not the same as a fixed line. A collineation in which every point and every line is fixed is the *identical* collineation. A *projective collineation* is a collineation where the mapping is a projective transformation.

Collineations form a group. Suppose the plane is transformed by a collineation α, and then by a collineation β. Each point P is mapped to $(P^{\alpha})^{\beta}$. There exists a collineation, called $\alpha\beta$, that performs the same transformation as the collineations α and β applied sequentially. If we define the operation of the group to be the *composition* of collineations, then we have closure. As well, the composition of collineations is associative. Each collineation α has an inverse, and we have an identity (the identical collineation). Thus collineations form a group.

A *central* collineation fixes every point on a line l and every line through a point V. Thus the line l and the point V are also fixed. No other lines or points in the plane are fixed. This is denoted as a (V, l) collineation, and V and l are the *centre* and *axis* of the collineation, respectively. A central collineation where V does not lie on l is known as an *homology*. If V lies on l, the central collineation is known as an *elation*.

For a given V and l, there are many (V, l) collineations. To specify a single (V, l) collineation α we must also give points P and P' such that $\alpha : P \to P'$. Furthermore, every projective collineation is a product of central collineations [1].

2.10 Polarities

Another type of transformation in the plane is the *correlation*. This is the dual of a collineation, in that it is a transformation of the plane that maps lines to points, points to lines and maintains incidences. Thus if σ is a correlation, and l and P are incident, then $\sigma(l)$ will be incident with $\sigma(P)$. The simplest and most useful correlation is the *dual* correlation. It maps lines and points with certain coordinates to points and lines with the same coordinates. That is it simply reverses the interpretation of coordinates. Homogeneous coordinates are discussed in Section 2.12.

A correlation of period two is known as a *polarity*. Given a polarity σ, $\sigma(P)$ is known as the *polar* of P, and $\sigma(l)$ is known as the *pole* of l.

2.11 Conics

Let τ be a projective transformation (not a single perspectivity) that takes lines through a point A to lines through a point B, where $A \neq B$. For all lines l through A, the point of intersection between l and $\tau(l)$ form the points of a conic section. This is how conics are defined in synthetic planar projective geometry.

Figure 2.13 shows two points A and B and a projectivity (defined by m, n, and O) that takes lines of A to lines of B. One such line l and its corresponding $\tau(l)$ is shown. The intersections of several corresponding pairs of lines is shown. As can be seen, they trace out an ellipse.

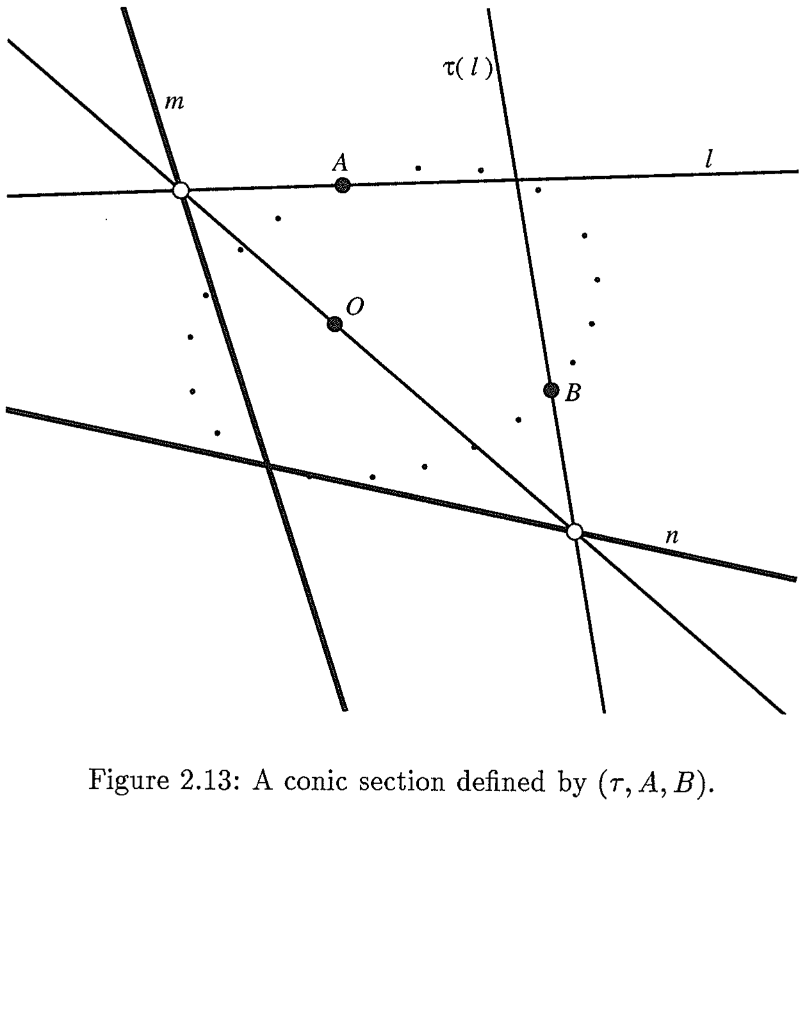

Figure 2.13: A conic section defined by (τ, A, B).

2.12 Homogeneous Coordinates

Up until now, our discussion of the projective plane has centred around *Synthetic* projective geometry. That is, the study of projective geometry that involves lines and points as they might be drawn on paper. If we wish to deal with *Analytic* projective geometry, we will need some way to represent projective lines and points algebraically. With this in mind, I will now introduce *Homogeneous Coordinates.*

Both points and lines are represented as triples of real numbers, with not all three being zero. To distinguish between them, points will have their homogeneous coordinates enclosed in parentheses and lines will have their homogeneous coordinates enclosed in square brackets. For example, the point P will have homogeneous coordinates (p_1, p_2, p_3), and line l will have homogeneous coordinates $[l_1, l_2, l_3]$.

Two triples with different homogeneous coordinates may refer to the same point or line. Specifically, if $P = (p_1, p_2, p_3)$ and $Q = (q_1, q_2, q_3)$ are two points with different homogeneous coordinates, we say that P and Q refer to the same point if there exists some real constant α such that $p_1 = \alpha \cdot q_1, p_2 = \alpha \cdot q_2, p_3 = \alpha \cdot q_3$. So a point or line can be considered a one-dimensional subspace of a 3-space.

For example, if $P = (3, 6, 5)$ and $Q = (6, 12, 10)$, we can show that P and Q in fact refer to the same point by taking $\alpha = 2$. In this case, P and Q are said to be two *representatives* of the same point.

To determine if a line and a point are incident, we calculate the dot product of their homogeneous coordinates. The point and line are incident if and only if the dot product is equal to zero. As well, three points are collinear (or three lines coincident) when the determinant of the matrix whose rows are their homogeneous coordinates is equal to zero. Therefore, if points P, Q and S are collinear, then there exists a linear dependence that holds, say $P = \alpha Q + \beta S$. In the next section is is shown that triples of numbers with this definition of incidence satisfy the axioms of the projective plane.

It is now possible to find a line that contains a pair of given points, or a point at the intersection of two lines. Given two points, $P = (p_1, p_2, p_3)$ and $Q = (q_1, q_2, q_3)$, we begin by representing the line containing them by the unknown $l = [l_1, l_2, l_3]$. The triple that results from calculating the cross product $P \times Q$ will be the homogeneous coordinates of the line containing P and Q.

To show this, let $l = P \times Q$. Thus, $l_1 = p_2q_3 - p_3q_2, l_2 = p_3q_1 - p_1q_3, l_3 = p_1q_2 - p_2q_1$. Now calculate the dot product of l and P.

$$\begin{aligned} l \cdot P &= p_1(p_2q_3 - p_3q_2) + p_2(p_3q_1 - p_1q_3) + p_3(p_1q_2 - p_2q_1) \\ &= p_1p_2q_3 - p_1p_3q_2 + p_2p_3q_1 - p_1p_2q_3 + p_1p_3q_2 - p_2p_3q_1 \\ &= 0 \end{aligned}$$

If S is another point, then S is on l if and only if $S \cdot l = S \cdot (P \times Q) = 0$. This is equivalent to the collinearity test mentioned earlier since $S \cdot (P \times Q)$ is the determinant of P, Q and S.

It should be pointed out that the elements of homogeneous coordinates need not be real numbers. Projective planes can be made with homogeneous coordinates based on rational numbers, complex numbers, as well as the elements of any finite field. In the rest of this thesis we will use real numbers, approximated by floating point numbers in the application implementation.

2.12.1 The Validity of Homogeneous Coordinates in the Projective Plane.

I will now prove the validity of homogeneous coordinates by showing that points and lines represented by homogeneous coordinates obey the axioms of planar projective geometry.

- *Any two points are contained in one and only one line.*

Firstly, it is clear that, given two distinct points $P = (p_1, p_2, p_3)$ and $Q = (q_1, q_2, q_3)$, where their components are real numbers, their cross product exists. So at least one line containing P and Q exists.

Any line containing P and Q must satisfy the following system of homogeneous equations:

$$l_1p_1 + l_2p_2 + l_3p_3 = 0$$

$$l_1q_1 + l_2q_2 + l_3q_3 = 0$$

where $l = [l_1, l_2, l_3]$ is unknown. It is clear that the coefficient matrix for this system of homogeneous equations has rank two, and that the system therefore has an infinite number of solutions. Since we have three unknowns (l_1, l_2, l_3), the solution space will have dimension one. Therefore, the solutions are all representatives of the same line, and this is the only line containing P and Q.

- *Any two distinct lines have in common one and only one point.*

  This argument is similar to the one given for the first axiom.

- *There exist at least for points, no three of which are contained in one line.*

  For this proof I will simply give the homogeneous coordinates of four points, no three of which have determinant zero. (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 1).

2.12.2 Visualizing homogeneous coordinates.

So far we haven't discussed how homogeneous coordinates relate to points and lines in the projective plane. The relationship between specific points and lines and their homogeneous coordinates is not defined. It is up to us to determine this relationship. The relationship is defined as a *visualization model* of homogeneous coordinates.

One model of visualization, indeed the one used in the application described by this thesis, is based on the idea that the homogeneous coordinates can be used to represent objects in Euclidean three-space. A projective point P with homogeneous coordinates (p_1, p_2, p_3) is taken to represent the Euclidean line that contains the origin and the point (p_1, p_2, p_3). A projective line l with homogeneous coordinates $[l_1, l_2, l_3]$ is taken to represent the Euclidean plane containing the origin and having orthogonal vector (l_1, l_2, l_3). To these lines and planes in Euclidean three-space we add another plane, called the *viewing plane*, which can be any plane that does not contain the origin.

The Euclidean lines representing projective points either intersect the viewing plane or are parallel to it. If they intersect the viewing plane, the points of intersection on the viewing plane represent the original projective points. Any Euclidean lines that are parallel to the viewing plane represent points at infinity, and the direction of each line represents the direction of its point at infinity.

The Euclidean planes representing projective lines either intersect the viewing plane or are parallel to it. If they intersect the viewing plane, the lines of intersection on the viewing plane represent the original projective lines. A Euclidean plane that is parallel to the viewing plane represents a line at infinity.

Rotating the viewing plane about the origin will change the points that appear to be at infinity. The distance of the viewing plane to the origin can also be changed to increase or decrease the scale of the image on the viewing plane. The program uses these techniques to implement panning and zooming, respectively.

Recall from elementary linear algebra that, if the dot product of two vectors in three-space is equal to zero, then the vectors are perpendicular. If the two vectors are the coordinates of a Euclidean line and a Euclidean plane as described above, then the line will lie in the plane. If the line and plane intersect the viewing plane, the intersection will appear as a point on a line. Thus the incidence of a projective line and point will be visualized correctly.

Also, recall that the cross product of two vectors will be a vector that is perpendicular to both original vectors. Thus, the cross product of the coordinates of two Euclidean lines will be the coordinates of the plane that contains them both, and the cross product of the coordinates of two Euclidean planes will be the coordinates of their line of intersection. These will represent, respectively, the projective line containing two projective points and the projective point at the intersection of two projective lines.

2.12.3 An Alternate Visualization.

While the visualization described in the previous section allows us to see the points at infinity by changing the viewing plane, it does not allow us to see all of the points all of the time. This is because changing the viewing plane may cause points that were previously visible to now appear at infinity.

Another visualization can be realized by using half of the surface of a sphere centred at the origin, with antipodal points identified, as the viewing surface instead of a plane. Projective points that are represented by Euclidean lines appear as points on the surface of the sphere. Projective lines that are represented by Euclidean planes appear as great circles on the surface of the sphere. Now points that previously appeared to be at infinity are visible at the 'equator' of the viewing sphere. The only drawback of this method is that the projective lines appear as great circles on the viewing sphere, and may appear curved.

Chapter 3

The Determining Set

In this chapter we introduce *Determining Sets.* Determining sets are used in several ways in this thesis, most notably for the movement of constructions. This approach was suggested by David Kelly [4].

Any construction in the projective plane can be represented by an ordered pair (Σ, Π), where Σ is the set of lines and points in the construction and Π is the set of incidences between those lines and points. For example, the triangle in Figure 3.1 can be represented by the pair $(\{A, B, C, l, m, n\}, \{An, Am, Bl, Bn, Cl, Cm\})$. This representation is abstract in that no physical manifestation of the construction (i.e. as drawn on paper) need exist for it to be studied. If we did want to draw this construction, we would first have to assign positions to the lines and points, and then draw them. These positions must be chosen so that the incidences in Π are maintained. For example, we know that the line l is incident with the points B and C. Therefore we must choose positions for B, C, and l such that, when they are drawn on paper, the line l is indeed incident with the points B and C. This chapter is concerned with the issues of assigning positions to lines and points in a construction while maintaining the incidences of that construction.

If we were to draw the triangle in Figure 3.1 on a piece of paper, we would probably draw three points in arbitrary positions and then place the lines through the pairs of points so that the incidence requirements were met. Alternatively we could begin by drawing three lines in arbitrary positions and placing the points at the intersections of these lines. In each case, notice that the positions of some of the objects in Σ are made redundant by some of the incidences. For example, if in Figure 3.1 we know the positions of the points $\{A, B, C\}$, the positions of the lines are determined by virtue of the fact that they must obey the incidences in Π. In other words, since every line in this construction contains two points, and in each case the positions of those two points are known, then every line can be placed as the line containing the points with which it is incident. By the axioms of planar projective geometry, we know that these lines exist and that they are unique.

In this example, we know (by axiom) that there is only one line through the points A and B. From Π we also know that the line l passes through A and B. Therefore, if the positions of A and B are known, then the position of l is also known.

This illustrates the concept of *determined* objects. That is, if the positions of two points are known, the position of the line containing them is *determined.* Therefore, a line containing two points can always be determined by those points. However, what if a line doesn't contain exactly two points?

For example, consider the projective construction defined by the pair $(\{P, l\}, \{Pl\})$. In this case neither object can be determined from the other. However, their positions cannot *both* be arbitrarily chosen, since there is an incidence to be maintained. Therefore, we say that one object is *constrained* by the other. If the position of one of the objects, say the point P, is known then the position of the other object (l) can be chosen arbitrarily, *so long* as it is incident with (passes through) P.

Now consider the projective construction defined by the pair $(\{A, B, C, l\}, \{Al, Bl, Cl\})$. If the positions of two of the points, say A and B, are known, then the position of line l can be determined. Once l is determined, it can be used to determine other points. However, since l is the only line passing through C, C can only be constrained by l.

This last example illustrates two important aspects of the determination of objects. First, once an object has been determined it can be used to determine other objects. Second, objects of one type (points or lines) can only be used to determine objects of the other type. In other words, points determine lines, and lines determine points.

We must now give a more precise definition of what it means to be determined. Let S be a set of points and lines. Let $S_0 = S$. We now define S_i, $i \geq 1$, to be

$$\begin{aligned} S_i = S_{i-1} &\cup \{P \mid P \text{ is incident on exactly 2 lines of } S_{i-1}\} \\ &\cup \{l \mid l \text{ is incident on exactly 2 points of } S_{i-1}\}. \end{aligned}$$

Let $S^* = S_j$ where $j = \min\{j \mid S_j = S_{j+1}\}$. All objects $o \in S^*$ are said to be *determined* by S. As well, the *rank* of an object o with respect to S is defined as $\min\{i \mid o \in S_i\}$. If an object has rank i, then it is said to be determined by S_{i-1}.

We are now ready to define *Determining Sets.* Given a construction (Σ, Π) in the projective plane, a determining set of this construction, denoted $\Delta(\Sigma, \Pi)$, is defined as a minimal subset of Σ that *determines* all of the objects in Σ, such that no two objects of the same rank are incident. That is, $\Delta(\Sigma, \Pi)$ determines all of Σ, but no proper subset of $\Delta(\Sigma, \Pi)$ determines all of Σ. Thus if the positions of the lines and points in $\Delta(\Sigma, \Pi)$ are known, then the positions of lines and points in $\Sigma - \Delta(\Sigma, \Pi)$ are also known. Some constructions do not have determining sets.

If, for a construction (Σ, Π), S is such that $S^* = \Sigma$, and no two objects of the same rank are incident, and S is minimal, then S is a determining set for the construction.

Note the restriction that no two objects with the same rank be incident. This is to prevent the case where two independantly determined objects are required to be incident, since this will not be true in general.

We will now clarify the process by which the positions of determined objects are computed. Recall that objects in the determining set, which have their positions defined, have rank 0. When using the determining set to find the positions of objects, the positions must be computed in order of increasing rank. Thus, we are given the positions of objects in the determining set, which have rank 0. We then compute the positions of objects of rank 1, then of rank 2, and so on.

Notice that an object o is only added to some S_i if it is incident on exactly two objects in S_{i-1}. Those are the two objects that determine o. When we come to determine the position of o, we know that the positions of its determining objects in S_{i-1} are known because they have lower rank and have therefore already had their positions computed. Note that, since the positions of our objects are stored as homogeneous coordinates (see Section 2.12), we can compute the position of o as the cross product of the positions of its two determining objects.

3.1 An Example with a Determining Set

Consider the construction shown in Figure 3.2. Suppose we had a set $S = S_0 = \{P_1, P_3, P_4, l_5\}$ and wanted to find the corresponding S^*. Notice that l_1 is incident on both P_1 and P_3, and is therefore determined by those two points. Since those two points are in S_0, we add l_1 to S_1, and say that l_1 has rank 1. Similarly, l_2 and l_4 are each incident on two points in S_0, and therefore can also be added to S_1.

Now notice that P_2 and P_5 are both incident on two objects from S_1, and can therefore be added to S_2. (In both cases one of the determining objects is l_5, which has rank 0.) The lines l_3 and l_6 are each incident on two objects of S_2, and can therefore be added to S_3.

We now compute S^* and find that it contains every object in the construction and therefore determines the construction. In addition, no proper subset of S determines the entire construction, and no two objects from any S_i are incident. Therefore, S is a determining set for the construction in Figure 3.2.

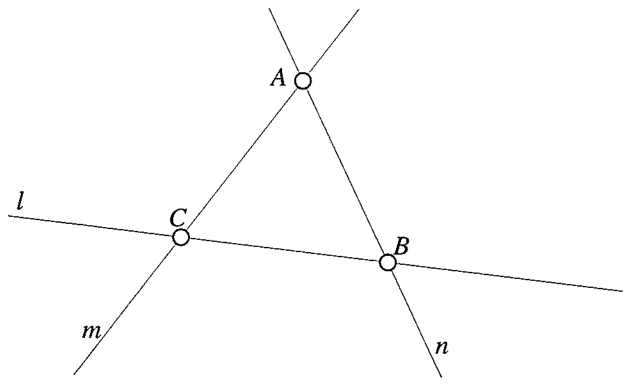

Figure 3.1: A Triangle ABC.

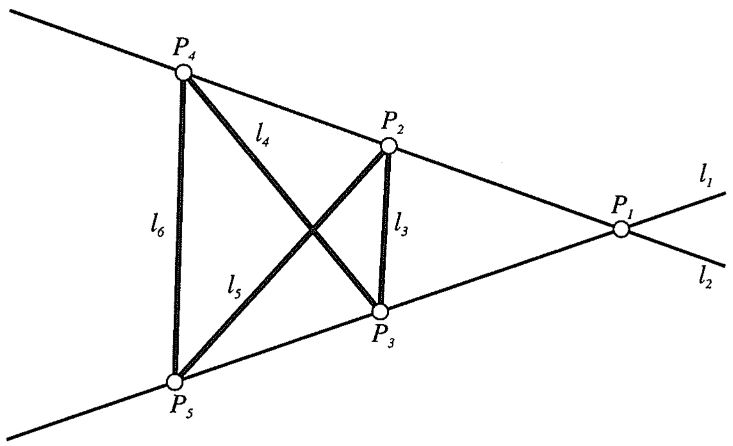

Figure 3.2: A Projective Construction.

3.2 General Position

Consider the following restrictions on the points and lines in the determining set as a consequence of the definition of determined. For a projective construction (Σ, Π), we cannot have three points $A, B, C \in \Sigma$ in the determining set if the incidences $lA, lB, lC \in \Pi$ for some $l \in \Sigma$. This is because l will be adjacent to *more* than two objects of lesser rank, which is a contradiction of the definition of determined.

Similarly, we cannot have three lines $l, m, n \in \Sigma$ in the determining set if the incidences $lA, mA, nA \in \Pi$ for some point $A \in \Sigma$.

We will now define *general position* as being a geometric constraint on the arrangement of lines and points. A set of points is said to be in general position if no three points in the set are collinear. A set of lines is said to be in general position if no three lines in the set are concurrent.

Notice the similarity between the concept of general position and the restrictions on points and lines in the determining set. Namely, that no three points can be in a determining set if there is a line containing them in the construction, and that no three lines can be in a determining set if there is a point containing them in the construction. In both cases we cannot have three points on a line or three lines through a point.

The difference is that general position refers to the positions of only lines or only points, where as with determining sets we are concerned with the abstract incidence structure of the lines and points.

We can extend the definitions of collinear and concurrent so that they apply to the incidence structure of a given construction. Suppose we are given a construction (Σ, Π). A set of points $P_1, P_2, \ldots, P_n \in \Sigma$ is said to be collinear with respect to the construction, if there is a line $l \in \Sigma$ such that the incidences $lP_1, lP_2, \ldots, lP_n \in \Pi$. Similarly, a set of lines $l_1, l_2, \ldots, l_n \in \Sigma$ is said to be concurrent with respect to the construction, if there is a point $P \in \Sigma$ such that the incidences $l_1P, l_2P, \ldots, l_nP \in \Pi$. Notice that points which are collinear in the plane need not be collinear with respect to a construction (Σ, Π). Using these definitions of collinear and concurrent, we can say that the points and lines in the determining set $\Delta(\Sigma, \Pi)$ are in general position.

3.3 Determining Sets and Incidence Graphs

Given a construction in the projective plane, we know we can derive its incidence graph. For example, the construction in Figure 3.2 has the incidence graph shown in Figure 3.3. Recall that the incidence graph of a projective construction contains one node for each line and point, and an edge representing each incidence of a point on a line. Suppose we are given a construction which has a determining set. We can now label the incidence graph to indicate the rank of each object, with objects in the determining set having rank 0. Having done this, we notice that all edges connect an object of lower rank with an object of higher rank. In fact, in each case the object with lower rank is used to determine the object with higher rank. In other words, each edge connects an object that is determined to another object determined by the first object.

This property of the edges must hold for constructions which have determining sets. To show this, consider all the situations an edge might have:

- An edge that connects two objects, neither of which is determined in terms of the other. These objects must have different rank, since no objects with the same rank can be incident. The object with higher rank is incident on at least three objects of lower rank. This cannot be because each determined object is incident with exactly two determined objects with lower rank (its determining objects).
- As well, if an edge connects two objects of different rank, the object of lower rank must determine the object of higher rank.

Therefore, an edge in the incidence graph of a determined construction must connect a determined object with another object determined by the first.

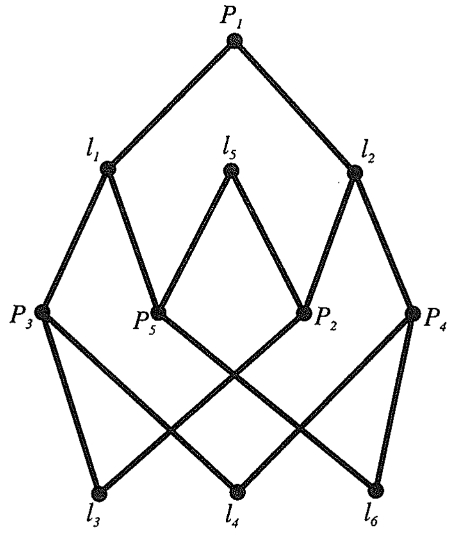

Figure 3.3: Incidence Graph for above Construction.

We can use this fact to discern several other interesting results about projective constructions with determining sets.

Proposition 3.3.1 *Given a projective construction* (Σ, Π), *and its determining set* $\Delta(\Sigma, \Pi)$, *it is possible to arbitrarily choose the positions of the objects in* $\Delta(\Sigma, \Pi)$ *and still maintain the incidences in* Π.

Proof In order to show that incidences are maintained, it is sufficient to show that no conflicts arise when the objects are placed at their assigned positions. A conflict is considered to exist when a point P and a line l in Σ have an incidence between them in Π, yet are not physically incident when the objects are place at their assigned positions.

The proof will be by induction on the rank of an object. Objects with rank 0 have no incidences between them, by definition of the determining set. Therefore we are completely free to arbitrarily choose their positions without fear of conflicts.

Assume that there are no conflicts among the objects in S_i. An object o of rank $i+1$ is incident on exactly two objects with rank less than $i+1$, with at least one having rank i.

These objects determine o. Recall that the position of a determined object is calculated as the cross product of the positions of its determining objects. For example, if the line l is determined by points A and B, in terms of homogeneous coordinates, $l = A \times B$. Therefore $l \cdot A = 0$ and $l \cdot B = 0$, and l is guaranteed to be incident with its determining objects. Thus there are no conflicts in S_{i+1}.

By induction we know that we can arbitrarily assign positions to the objects in the determining set and thus determine the positions of all objects in the construction without contradicting the incidences of Π. $\square$

Proposition 3.3.2 *Given a construction which has a determining set, its corresponding incidence graph will have an even number of edges.*

Proof Since every determined object is determined by two other objects, and since every edge in the incidence graph connects a determined object to an object it determines, we know that there must be an even number of edges in the incidence graph for this construction. Therefore, for a construction to have a determining set, it must have an even number of edges in its incidence graph. $\square$

Note that the condition that the incidence graph for a construction must have an even number of edges for the construction to have a determining set is necessary, but not sufficient. For example, the Desargues' configuration, which has 30 edges in its incidence graph, has no determining set. This is because, as we shall see later, it is a (3,3)-configuration.

Proposition 3.3.3 *Given a construction with a determining set and an incidence graph G. Let E be the number of edges in G, let n be the number of nodes (objects) in G, and let k be the number of objects in the determining set. Then $k = n - E/2$.*

Proof Since the construction has a determining set, we know that every line and point in the construction is either in the determining set, or determined by the determining set. Therefore we know that there are k objects in the determining set and $n - k$ objects determined by the determining set. Since each determined object has two edges associated with it, there must be $2(n - k)$ edges in G. If we know the values of E and n, we can solve for k. Thus, if the determining set for a given construction exists, it will contain $n - E/2$ objects. □

Since there may be more than one determining set for a construction, we can also say that all determining sets for a particular construction must contain the same number of objects. We say that this is the *dimension* of a construction. For example, the construction in Figure 3.2 has dimension 4.

3.4 An Algorithm to Find Determining Sets

We will now give an algorithm for finding a determining set of a projective construction, (Σ, Π). Assume that we have an empty stack Q, and an empty set S. The algorithm executes the following loop until it either finds a determining set or until it determines no determining set exists:

- Select an object o from Σ that is not incident with any object in Q. Add o to S and push it onto Q.
- Find all undetermined objects in Σ that can now be determined by the objects in S and push them onto Q. If in this step we find an undetermined object that is adjacent to more than two determined objects, then we perform a backtrack.

The second step continually iterates through the set of undetermined objects in Σ until the set remains stable for two consecutive iterations. In each iteration it searches the set for objects which are adjacent to exactly two objects from Q. These objects can now be determined from S. If one of the undetermined objects is adjacent to more than two objects from Q, we perform a backtrack. A backtrack returns Q and S to the state they were in just before the last object was added to S.

Since we are using a backtracking algorithm, it is important that, after performing a backtrack, we select a different object o than that which caused the backtrack. We also need a way to detect when all possible variations of S have been tried, in which case no determining set exists for Σ.

3.5 Augmented Determining Sets

There are some projective constructions for which no determining set exists. For example, Figure 3.4 shows two triangles in perspective from a point. This construction has no determining set. We know from Section 3.3 that, if a determining set exists for a given construction, its incidence graph must have an even number of edges. Now consider the incidence graph for the construction in Figure 3.4, shown in Figure 3.5. This graph has 21 edges, so therefore the construction cannot have a determining set. □

We now introduce augmented determining sets. Briefly, augmented determining sets are the same as determining sets except that they may contain constrained objects. Let (Σ, Π) be a projective construction which has no determining set. Let $S \subseteq \Sigma$ be a set of lines and points that is a determining set for a construction (Σ', Π'), where $\Sigma' \subset \Sigma$, and $\Pi' \subset \Pi$, and for every incidence $lP \in \Pi'$, $l, P \in \Sigma'$. While S does not determine the entire construction, there is a set of determined objects S^* corresponding to S. Let o be an object of $\Sigma - S^*$ which is incident on exactly one object $m \in S^*$, if there is such an object o. The object o is constrained by m. Assume that m has rank i. We then add o to S_{i+1} and recompute S_k, for $k > i + 1$, since the addition of o may allow more objects to be determined.

Let S' be a set of constrained objects. If $\{S \cup S'\}^* = \Sigma$, and $\{S \cup S'\}$ is minimal, then $\{S \cup S'\}$ determines the drawing. These sets will be called *augmented determining*

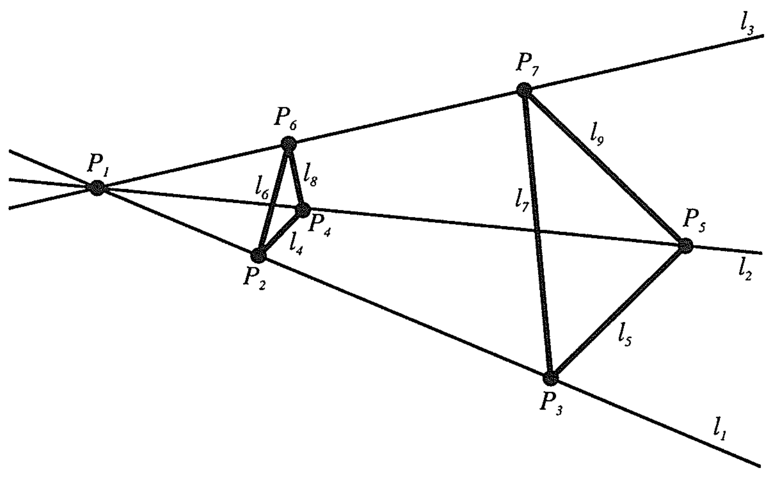

Figure 3.4: Two Triangles in Perspective.

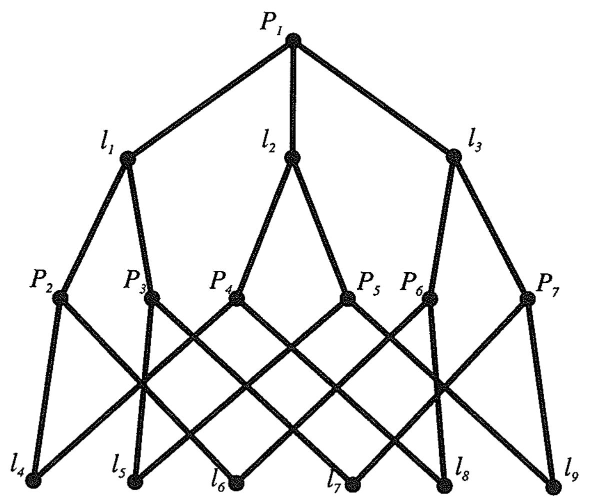

Figure 3.5: Incidence Graph for Two Triangles in Perspective.

sets, and will be denoted by $\Delta'(\Sigma, \Pi)$. In section 3.5.1 we will see how to compute an augmented determining set with an example.

Note that the objects in S' can be constrained on objects of rank zero or on objects with rank greater than zero.

The reason we introduce augmented determining sets is that they will allow us to determine some constructions that could not be determined with determining sets alone. In fact, it is possible to find an augmented determining set that determines all objects in the construction in Figure 3.4.

One of the consequences of allowing constrained objects in a determining set is that we are no longer free to arbitrarily chose the positions of all the objects in a determining set. The constrained objects must be placed such that they are incident with their constraining objects.

Note that there are still some constructions for which no determining set or augmented determining set exists. For example, any (3,3)-configuration has no augmented determining set.

This can be proven as follows. Suppose that we are given a (3,3)-configuration which *does* have a augmented determining set. This implies that every object in the configuration is determined. Every node in the incidence graph for this configuration will have degree 3. Recall that each edge in the incidence graph of a determined construction must connect a determined object to another object determined by the first.

In this case, every object in the configuration falls into one of three categories:

- An object in the determining set. This object is determined and in turn can be used to determine three other objects. In the incidence graph this object supplies three edges that can be used to determine other objects.
- A constrained object in the augmented determining set. This object is determined and in turn can be used to determine two other objects. In the incidence graph this

object is constrained using one edge from a determined object, and supplies two edges that can be used to determine other objects.

- An object determined by two other objects. This object is determined and in turn can be used to determine one other object. In the incidence graph this object is determined using two edges from determined objects, and supplies one edge that can be used to determine other objects.

Note that, in all of these cases, the object is determined and provides at least one edge in the incidence graph for determining other objects. Since, by hypothesis, every object is determined, there will be at least one edge in the incidence graph that is *left over.* This edge connects two determined objects, yet neither of these objects are determined in terms of the other, and it cannot be ignored by using a constraint. This is a contradiction, therefore this configuration has no determining set or augmented determining set. □

3.5.1 An Example with an Augmented Determining Set

In the previous Section, we claimed that there existed an augmented determining set that determined the construction in Figure 3.4. Using the labeling of Figure 3.5, we will now show the construction of this augmented determining set as an example.

Suppose we began by trying to construct a non-augmented determining set S. Let S contain initially the point P_1. Since S does not yet determine the construction, we add objects to S. Knowing the restrictions on the objects in a determining set, we cannot select any of l_1, l_2, or l_3. We therefore add P_2, P_4, and P_6 to S. At this point, $S^* = P_1, P_2, P_4, P_6, l_1, l_2, l_3, l_4, l_6, l_8$. Note that, at this point, the points have rank 0 and the lines have rank 1. Since this still does not determine the entire construction, we must add more objects to S. We cannot add any more points because all the remaining points are adjacent to determined lines. If we did add these points then we would have lines in S^* that were adjacent to more that two objects of lesser rank, which is illegal. Let us then add l_5 to S. This will allow us to determine points P_3, P_5 with rank 2.

At this point we reach a dead end. All remaining undetermined objects are adjacent to determined objects, so we can not add anything else to S. However, there are still undetermined objects. We cannot find a non-augmented determining set, so we now begin expanding S into an augmented determining set. Let us add l_9, which is constrained by P_5, to S'. Since P_5 has rank 2, l_9 will have rank 3. We can now determine P_7, with rank 4. This will enable us to determine l_7, having rank 5.

Since $\{S \cup S'\}^*$ contains all objects in the construction, it is an augmented determining set for the construction.

3.5.2 Augmented Determining Sets and Incidence Graphs

Recall from Section 3.3 that every edge in the incidence graph of a construction having a determining set must connect a determined object to an object it determines. However, if we allow constrained objects in the determining set, as we do with augmented determining sets, then this result no longer holds.

For example, let l and P be a line and a point in the augmented determining set for a construction (Σ, Π), such that the incidence $lP \in \Pi$. Thus l and P are a constrained pair of objects in the augmented determining set. In the incidence graph for this construction, the edge corresponding to the incidence lP will connect two objects, neither of which is determined in terms of the other. Even though every object in the construction is determined, this edge exists and our previous result is contradicted. We must therefore derive a new result for constructions having augmented determining sets.

The reason we previously disallowed edges like the one described above was because, in general, we cannot arbitrarily choose positions for l and P such that the incidence lP is maintained. However, since l and P are a constrained pair of objects in the augmented determining set, we know that their positions will be chosen such that the incidence lP will be maintained. The existence of this edge in the incidence graph does not cause any conflicts when positions are assigned to the objects in the augmented determining set, and is therefore valid. In other words, for augmented determining sets, we make the distinction between edges that are used to determine objects and edges that do not determine objects, with the latter corresponding to constrained pairs.

We can now say that an edge in the incidence graph of a construction having an augmented determining set connects a determined object to an object it determines if and only if that edge does not correspond to a constrained pair of objects.

Recall that, for a construction having a determining set, its incidence graph G had to satisfy the condition $E = 2(n - k)$. We can now derive the corresponding result for constructions having augmented determining sets.

Proposition 3.5.1 *Given an incidence graph G for a construction having an augmented determining set Δ', let E be the number of edges in G, let n be the number of objects in G, let k be the number of* non-constrained *objects in Δ', and let c be the number of* constrained *objects in Δ'. If Δ' exists, then the equation $2k + c = 2n - E$ must hold.*

Proof Notice that each constrained object has an edge that connects it to its constraining object. Even though the constrained objects help to determine other objects, this edge is not used to determine any object.

Therefore, the number of edges that are used to determine objects is $E - c$. Each object that is determined requires two such edges. Since there are $n - k - c$ determined objects, we can say that $E - c = 2(n - k - c)$. It then follows that $2k + c = 2n - E$. □

Notice that, since n and E are constant for a given construction, $2k + c$ is an invariant for that construction, regardless of the augmented determining set chosen.

We can now see how the use of augmented determining sets can help to determine constructions that don't have determining sets. Recall that the construction in Figure 3.4 has 21 edges in its incidence graph and therefore has no determining set. We claim that this construction has an augmented determining set. Consider an augmented determining set Δ' having one constrained object. Now the number of edges that determine objects is equal to $E - c = 21 - 1 = 20$. Since 20 is even, it may now be possible to use all of these edges to determine objects.

Recall that the dimension of a construction having a determining set is the number of objects in the determining set. We now define the dimension of a construction having an augmented determining set Δ' as being the number of *non-constrained* objects in Δ' plus half the number of *constrained* objects in Δ', or $k + c/2$. For example, the construction in Figure 3.4 has dimension 5 1/2.

Chapter 4

Movement

In this chapter we will discuss the problem of moving the objects in a construction while maintaining its incidence structure. Suppose we are given a projective construction (Σ, Π) which has a valid coordinatization. That is, homogeneous coordinates have been assigned to the objects in Σ such that the incidences in Π are maintained. Now suppose that we wish to change the appearance of the construction by moving a point $P \in \Sigma$ by changing its coordinates. Moving P, however, can introduce conflicts in the coordinatization of the rest of the construction. For example, there may be one or more lines $l \in \Sigma$ such that $lP \in \Pi$, but for the new coordinates of P, $l \cdot P \neq 0$. Thus we will also have to move all such lines l. But this may introduce conflicts between these lines and other points incident with them, causing us to move more points to maintain the incidences of Π. Moving these points may cause conflicts with other lines incident on them, and so on.

For example, consider the triangle in Figure 3.1. Suppose we move the point A such that it is no longer on lines m and n. We now have to move lines m and n so that they are once again incident on A. If the movement we apply to these lines also moves them off of points C and B, then we will also have to move those points. But the movements we apply to the points may move them off of the line l. We will then have to move the line l back onto the points C and B.

We can see that moving one object in a construction can have far reaching side effects on the rest of the construction. It is also not clear what form the side effects will take. In the above example, we moved the line m to maintain its incidence with the point A. There are, however, an infinite number of movements we can apply to m that will move it onto A. Without further selection criteria, the choice of movement becomes ambiguous.

Here we will discuss two methods for dealing with these problems. The first uses determining sets as discussed in Chapter 3. The second method uses central collineations, first introduced in Section 2.9.

4.1 A Method using Determining Sets

Determining sets were developed to help find valid coordinatizations for constructions. By arbitrarily assigning coordinates to the objects in the (un-augmented) determining set for a construction, a valid coordinatization can be found for the entire construction. In the application developed for this thesis, however, this is not an issue. The user enters constructions by specifying the positions of the lines and points that make up the construction. Thus a valid coordinatization is always available for the construction being manipulated.

Determining sets are used, however, to solve the problems that arise when an object in a construction is moved. Consider a construction (Σ, Π) which has a valid coordinatization. Suppose we want to move a point $P \in \Sigma$. Let Δ be a determining set for this construction containing the point P. We can now apply any movement to P and, leaving the other objects in Δ fixed, simply recalculate the positions of the rest of objects using the technique described in Chapter 3.

One problem that arises from this application of determining sets is that the nature of the movement produced is greatly affected by the determining set being used. Determining sets are, in general, not unique. That is, for a given construction, several different but equivalent determining sets may exist. Thus, for a given movement of an object in a construction, the movement of the rest of the construction will vary depending on the determining set used.

This becomes a problem when we consider that, in some cases, a user will want to move one object in a construction in order to see its effect on another object. If this object is fixed by the determining set, then the movement will be of no use to the user. As well, other algorithms discussed in Chapter 5 can have varying behavior depending on the determining set chosen.

What is needed is a way to constrain the selection of the determining set for a construction. This is accomplished by allowing the user to specify objects that are to be in the determining set. Since these objects will not be animated in the movement, this process is known as *fixing* objects in the construction. These objects are guaranteed to be in the determining set, while the remaining objects in the determining set are chosen by the determining set algorithm.

Note that the user may fix objects which make it impossible to find a determining set. In this case, no movement can be made until the user remedies the problem.

4.2 A Method using Central Collineations

Recall collineations from Section 2.9. A collineation is a transformation of the projective plane that maps points onto points and lines onto lines, while maintaining incidences. In the previous section we discussed a technique that used determining sets to move some of the objects in a construction while maintaining the incidences of the construction. This movement can be considered to be a transformation of the projective plane as mentioned above, which fixes an undefined number of objects and maintains incidences. By the same token, a collineation can be viewed as a movement of objects in the plane which maintains incidences. If this is a central collineation, then the movement keeps one point

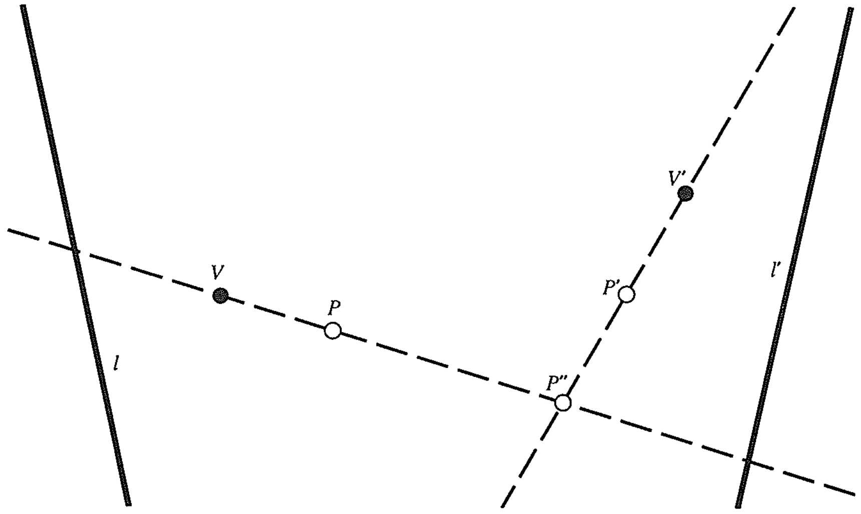

Figure 4.1: Moving a point with two Collineations.

of fixed lines fixed and one line of fixed points fixed.

Thus we can use central collineations to implement a new type of movement. Recall that specifying the centre and axis of a central collineation in fact describes a set of central collineations. To specify one collineation α in the set, we give points P and P' such that $\alpha : P \rightarrow P'$. Suppose, in our application, we have a construction that contains a point P, and that the user wishes to move it to P'. If we have a central collineation α such that $\alpha : P \rightarrow P'$, then this collineation will not only move P to the desired position, but it will also move other objects in the construction and maintain incidences.

Note that, given a centre V, the point P can only be moved along the fixed line VP. In order to allow a freer range of movement, we allow the user to define the centre and axis of two central collineations, (V, l) and (V', l'). Whenever the user moves a point P to P', we find the (V, l) collineation α and the (V', l') collineation β such that $(P^\alpha)^\beta = P'$. We then apply the collineation $\alpha\beta$ to the entire construction.

Figure 4.1 shows how α moves P to P'', and then β moves P'' to P'. Performing these transformations in this order on all other objects in the construction will perform the same collineation on the rest of the construction. This is further described in Chapter 6.

Chapter 5

Finding Incidences in General

We now consider the problem of assigning positions to objects in a projective construction in order to preserve incidences. Consider the projective construction (Σ, Π). Let Σ contain the points $\{P_1, P_2, \cdots, P_m\}$ and the lines $\{l_1, l_2, \cdots, l_n\}$. For every $l_iP_j \in \Pi$, we create an equation $l_i \cdot P_j = 0$. If there are k incidences in Π, then we will create k equations. These equations are a necessary and sufficient condition for the assigned coordinates to be valid.

Recall the description of the Desargues' configuration. Given two triangles which are in perspective, the intersections of corresponding sides of the triangles are collinear. The fact that the three points of intersection are collinear is inherent in the theorem which describes this configuration. In other words, we can take the line determined by two of the points of intersection, and we know that the third point of intersection lies on that line. However, given a projective construction that appears to contain a certain incidence, how can we determine if that incidence will hold in general?

Given a projective construction (Σ, Π) and a determining set $\Delta(\Sigma, \Pi)$, we know that we can arbitrarily choose the positions of the objects in the determining set, and the positions of the rest of the objects in the construction are uniquely determined. Suppose that P and Q are two points in the determining set, and that they determine the line l.

In terms of their homogeneous coordinates, we can say that $l = P \times Q$. If we move P and Q by a certain amount, say δ_P and δ_Q respectively, the new coordinates l' of l are given by the equation $l' = (P+\delta_P)\times(Q+\delta_Q) = P\times Q \ + \ P\times\delta_Q \ + \ \delta_P\times Q \ + \ \delta_P\times\delta_Q$. Since $l = P \times Q$, this equation becomes $l' = l \ + \ P\times\delta_Q \ + \ \delta_P\times Q \ + \ \delta_P\times\delta_Q$. If we now define δ_l to be $\delta_l = P\times\delta_Q \ + \ \delta_P\times Q \ + \ \delta_P\times\delta_Q$, we can say that $l' = l + \delta_l$. Now, given the changes in the position of the objects in the determining set Δ, we can compute the changes in the position of the rest of the objects in the construction.

Suppose we have the positions of objects in the determining set of a construction (Σ, Π) which has an apparent incidence between a line m and a point S (i.e. $m \cdot S = 0$, but mS is not in Π). If we change the positions of the objects in the determining set, we can find the resulting changes in the positions of m and S, δ_m and δ_S respectively. In order to see if m and S are still incident, we must determine whether of not the equation $(m+\delta_m)\cdot(S+\delta_S) \ = \ m\cdot S \ + \ m\cdot\delta_S \ + \ \delta_m\cdot S \ + \ \delta_m\cdot\delta_S \ = \ 0$ holds. If this equation holds, then m and S remain incident after the given movement has been applied to the objects in the determining set.

This method can be extended to determine if an incidence holds in general. Suppose that, instead of supplying the changes in position of objects in the determining set as constants, we supply the changes in position as symbolic variables. If we carry these variables through our calculations, we will eventually arrive at changes in position of all objects in the construction in terms of these variables. If the equation $(m+\delta_m)\cdot(S+\delta_S) = 0$ now holds, we know that, for any given position of objects in the determining set, m and S will be incident.

Suppose we wanted to use this technique to illustrate Desargues' theorem. We could do this by drawing a construction consisting of two triangles in perspective, and the three points of intersection between corresponding sides. If we add the line determined by two of these points of intersection, we can hypothesize that this line will always be incident with the third point of intersection. Let this line be m and the third point of intersection be S. We want to show that the incidence mS holds in general.

We now find a determining set for this construction. Note that, since this is not the full Desargues' configuration, the determining set exists. We then assign coordinates to the objects in the determining set, and compute the coordinates of the rest of the construction. Computing the value of $m \cdot S$, we find that it is zero, and therefore m and S are indeed incident. We must now determine if this incidence holds in general or if it is just a coincidence of this particular coordinatization. Thus we assign variable movements to the objects in the determining set, and compute the movements of the rest of the object in the construction in terms of these variables. We then calculate $(m + \delta_m) \cdot (S + \delta_S)$ and find that it is also zero. Our hypothesized incidence is thus proven for all positions of objects in the determining set.

As well, the above technique shows us that, even though Δ is the determining set for a simplified Desargues' configuration, it maintains the incidences of the full Desargues' configuration, and can therefore be used to coordinatize the full Desargues' configuration. Thus, it may now be possible to coordinatize configurations which, in their full form, have no determining sets.

5.1 More Configurations

So far we have been introduced to two configurations, Pappus and Desargues'. We will now look at a few more configurations. The 7_3 configuration shown in Figure 5.1 is also known as the *Fano* configuration. As you can see this drawing of the Fano configuration contains a 'line' which is circular. In fact, this configuration cannot be drawn with the application developed for this thesis, or in the real projective plane [13]. Similarly, the 8_3 configuration in Figure 5.2 cannot be drawn in the real projective plane.

It may seem odd to consider these to be projective configurations, because they cannot be drawn with just straight lines and points. However, they do obey the three axioms

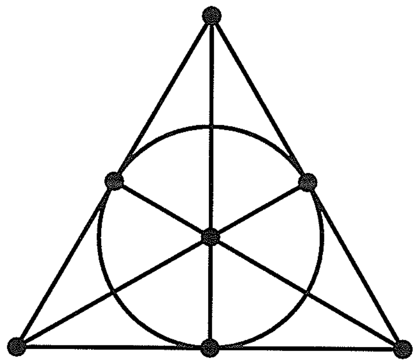

Figure 5.1: The 7_3 Fano Configuration

of planar projective geometry. In fact, the Fano plane, which is a finite projective plane over GF(2), can be used to coordinatize the Fano configuration.

The Pappus configuration is not the only 9_3-configuration. As Figure 5.3 shows, it is only one of three 9_3-configurations. Similarly, there are ten 10_3-configurations, of which Desargues' is only one [2]. The Desargues' configuration is a theorem in any projective plane coordinatized by homogeneous coordinates.

Of the three 9_3-configurations, only the Pappus configuratin is easy to coordinatize, since it is also a theorem. One simply follows the procedure in Section 2.7 and the three collinear points of intersection emerge. For the other two configurations, the incidence structure is not so easily achieved. We need to solve a system of equations to find coordinatizations that maintain the incidences of the configurations.

Similarly, of the ten 10_3-configurations, only Desargues' is easy to coordinatize. One of the configurations cannot be coordinatized over any commutative field, and the other eight require a solution to a system of equations.

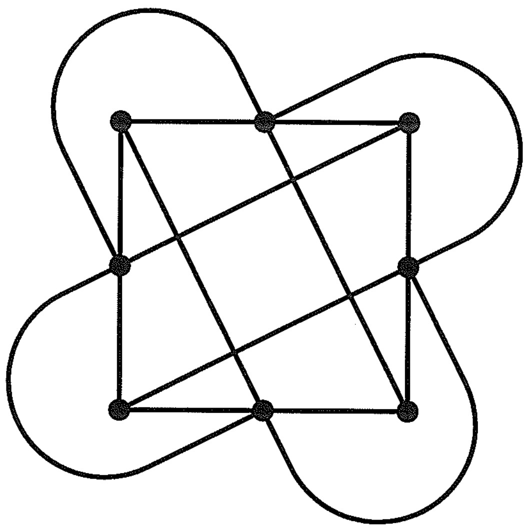

Figure 5.2: The 8_3 Configuration

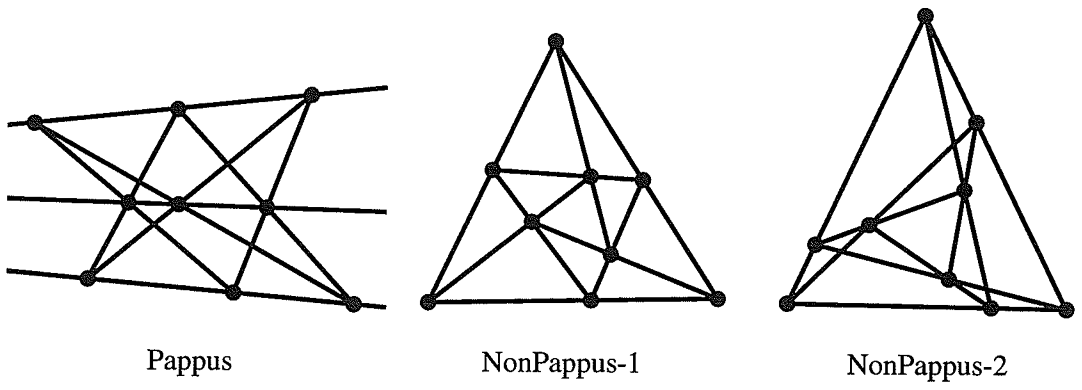

Figure 5.3: The Three 9_3 Configurations

5.2 Forcing Incidences

Suppose we are given a construction (Σ, Π) which has a valid coordinatization, and which contains a line l and a point P which are not as yet incident (i.e. $l \cdot P \neq 0$ and Π does not contain the incidence lP). One operation we may wish to perform is to *force* l and P to be incident. In other words, find a new coordinatization in which $l \cdot P = 0$, and add the incidence lP to Π.

For example, if we want to construct the non-Pappus 9_3-configurations, or the non-Desargues' 10_3-configurations, we can easily construct them with one incidence missing, and then force this last incidence to get a valid coordinatization.

Another reason why we may want to perform such an operation is that it may be possible to coordinatize (3,3)-configurations even though they have no determining sets. As we showed earlier in this chapter, this can be achieved by removing one incidence from the configuration and finding a determining set for the simplified configuration. If the determining set can be shown to maintain the incidence that was removed, as will happen with the Desargues' configuration, then that determining set can be used to move the simplified configuration as if it were the full configuration. However, what if the determining set for the simplified configuration did *not* maintain the incidences of the full configuration? For example, the two non-Pappus 9_3-configurations. If this were the case then there would be one incidence in the construction that is not maintained. This missing incidence, the incidence that was removed, would then have to be explicitly enforced.

We now describe a technique that can be helpful in forcing objects to be incident. Consider the construction (Σ, Π) mentioned at the beginning of this chapter. In order to force l and P to be incident, we would first have to find a coordinatization of the objects in Σ that maintains the incidences in $\Pi \cup \{lP\}$.

Let $\Delta(\Sigma, \Pi)$ be a determining set for this construction which contains P but not l.

Thus P will have rank 0 and l will be determined with rank greater than 0. As before, let us assign variable movements to the objects in Δ and calculate the resulting movements in the rest of the construction. Consider the value of the expression $(l+\delta_l)\cdot(P+\delta_P)$. Since l and P are not incident, this expression will be a non-zero function of the movements of the objects in Δ.

Now consider the equation $(l+\delta_l)\cdot(P+\delta_P)=0$. Notice that any solution to this equation will define the movements of the objects in Δ that will cause l and P to be incident. That is, after applying the solution (movements) to the objects in Δ, $l'\cdot P'=0$. Thus we only have to solve the above equation and apply that solution to the objects in Δ and the desired incidence will be achieved. We call $(l+\delta_l)\cdot(P+\delta_P)=0$ the *constraining equation*, because any movements for the objects in Δ must be constrained by it in order to be valid movements.

However, the constraining equation may be quadratic in a theoretically unlimited number of variables. It is not clear how we could solve such an equation. Therefore we must simplify this technique so that it will produce a constraining equation that we can solve in real time.

Notice that we currently assign variable movements to all objects in the determining set. Thus we are solving for movements of *all* objects in the determining set that will cause l and P to be incident. The larger the determining set, the more complex the equation. What if we only solved for the movement of one object in Δ, say P, and fixed the other objects in Δ? By assigning variable movement to only one object in Δ, we not only lower the number of variables we have to deal with, but we also ensure that we have a constant number of variables in the equation. Since we are trying to force the incidence of l and P, solving for the movement of P that will move it onto l makes sense. Since we are using homogeneous coordinates to represent positions, our variable movement for P will be represented by a variable homogeneous coordinate triple.

Now notice that the equation is quadratic in the δ's, and is therefore difficult to solve.

Instead of solving the quadratic equation, we linearize it by ignoring the quadratic terms. Thus, if $l = P \times Q$, then we take δ_l to be equal to $P \times \delta_Q + \delta_P \times Q$ and ignore the $\delta_P \times \delta_Q$ term. We also now take $(l + \delta_l) \cdot (P + \delta_P)$ to be equal to $l \cdot P + l \cdot \delta_P + \delta_l \cdot P$ and ignore the $\delta_l \cdot \delta_P$ term. This can be justified by noticing that, in most cases, δ_P will be smaller than P, and thus the quadratic terms will not contribute a great deal to the value of the equation. Having said this, it is important to note that any solution we get from the linearized equation will only be an approximation. We will have to use an iteration-based technique to get a result that will be suitable for our purposes.

Notice that, in the computer application, we are working with floating point numbers, which are an approximation of real numbers. As such, any computation we perform is already an approximation. Thus, using using an iteration-based technique to find an approximate solution is not significant.

We will now discuss the form of the constraining equation after the simplifications have been applied. First notice that each δ is calculated as the sum of two cross products. These are cross products of a homogeneous coordinate and another δ. The first δ's we calculate will involve objects in Δ. Suppose we calculate $\delta_m = (P + \delta_P) \times (S + \delta_S) = P \times \delta_S + \delta_P \times S$, where $P = (P_1, P_2, P_3)$ and $S = (S_1, S_2, S_3)$ are in Δ. Let $\delta_P = (x, y, z)$ be the symbolic movement of P, and let $\delta_S = (0, 0, 0)$, since S is fixed. Since δ_s is zero, the first cross product is zero. Thus $\delta_m = \left(\begin{vmatrix} y & z \\ S_2 & S_3 \end{vmatrix}, - \begin{vmatrix} x & z \\ S_1 & S_3 \end{vmatrix}, \begin{vmatrix} x & y \\ S_1 & S_2 \end{vmatrix} \right)$. As you can see, the components of δ_m will be linear combinations of x, y, and z. If δ_S were not equal to zero, it would also be made up of linear combinations of (x, y, z), and δ_m would again be made up of linear combinations of (x, y, z).

Now consider the constraining equation, $(l + \delta_l) \cdot (P + \delta_P) = l \cdot P + l \cdot \delta_P + \delta_l \cdot P = 0$. It is calculated as the sum of three dot products. Two of the terms are dot products involving one homogeneous coordinate and one δ. These simplify to linear combinations of (x, y, z). Let the sum of these terms be $rx + sy + tz$, where r, s and t are real numbers. The third term is the dot product of two homogeneous coordinates, and simplifies to a real numeric constant. Let this constant be $c = l \cdot P$. Our constraining equation thus simplifies to $rx + sy + tz + c = 0$.

We now give an algorithm that describes our simplified technique for forcing the incidence of a line l and a point P in a construction (Σ, Π). Since an iteration-based algorithm will not, in general, be able to take $l \cdot P$ *exactly* to zero, we will define $|l \cdot P| < \varepsilon$ to be the finishing condition. The value of ε chosen will determine how accurate the algorithm is.

- Find a determining set $\Delta(\Sigma, \Pi)$ which contains P but not l. Assign (x, y, z) as the movement for P, and $(0, 0, 0)$ as the movement for the objects in $\Delta - \{P\}$.
- While $|l \cdot P| < \varepsilon$ do
  - Calculate the δ's of the rest of the objects in the construction in terms of δ_P. These will be linear combinations of x, y, and z.
  - Evaluate the expression $(l + \delta_l) \cdot (P + \delta_P)$. Because of the simplifications, this will be of the form $rx + sy + tz + c$ for some real numbers r, s, t, and c.
  - Solve the constraining equation $rx + sy + tz + c = 0$ for x, y, and z.
  - Apply the movement (x, y, z) to P and recalculate the position of objects in the construction.
- Add the incidence lP to Π.

5.2.1 Methods for solving the constraining equation

In the algorithm above we must solve the equation $rx + sy + tz + c = 0$ to find the movement (x, y, z) for the point P. Notice that we have a tremendous amount of freedom in choosing (x, y, z). One simple method is to select values for x and y and solve for z. Thus $(x, y, z) = (x, y, \frac{-(c+rx+sy)}{t})$. To address the possibility that t might be zero, we would simply solve for another of the three variables.

The problem with the above method is that the movements we find with it are limited by our choice of x and y. Remember that these movements are to be applied to a construction that the user is altering in real-time. Thus the movements of the objects should be continuous and not erratic. If our choice of x and y is inappropriate, the objects may move erratically. Another way of looking at this problem is that only one of the three variables is affected by the constraining equation. Thus the constraining equation does not have very much effect on the movement we choose.

Consider the movement $(x, y, z) = (\frac{-c}{3r}, \frac{-c}{3s}, \frac{-c}{3t})$. This movement satisfies the constraining equation. As well, the effect of the constraining equation is more evenly spread across all three components of the movement than with the previous method. Also notice that $c = l \cdot P$ is a measure of how close l and P are. Thus, as l and P move closer to each other with each iteration, c becomes smaller and the movement applied becomes smaller. This helps to reduce the oscillation that can occur with iteration-based algorithms.

This method still has a problem. It does not take into account the current position of P. Thus, applying the movement to P could drastically change the position of P and cause erratic movement in the rest of the construction.

Let $P = (P_1, P_2, P_3)$ be the homogeneous coordinate of the current position of P. Consider the movement $(x, y, z) = (\frac{-c \cdot |P_1|}{(|P_1|+|P_2|+|P_3|)r}, \frac{-c \cdot |P_2|}{(|P_1|+|P_2|+|P_3|)s}, \frac{-c \cdot |P_3|}{(|P_1|+|P_2|+|P_3|)t})$. This movement again satisfies the constraining equation, but also takes into account the current position of P. Thus the larger components of P will be changed by a greater amount than the smaller components.

When these methods were tested, the last method was indeed found to produce the *nicest* and most consistent results.

5.3 Movement in Constructions without Determining Sets

Suppose we are given a (3, 3)-configuration (Σ, Π), with valid coordinates, and wish to alter its appearance by moving one of its objects. Let o be the object being moved. Normally we would find a determining set $\Delta(\Sigma, \Pi)$ containing o, and then simply recalculate the positions of the objects every time o is moved. However, as we know from Chapter 3, a (3, 3)-configuration has no determining set, so we cannot use this method of object-movement. In this case, we select an incidence $lP \in \Pi$ and remove it from Π. We then check to see if the incidence lP holds in general. If the incidence *does* hold in general, then we try to move the simplified construction. Note that, after removing the incidence, the construction is no longer a (3,3)-configuration, and therefore may have a determining set.

But what happens if the incidence lP is not true in general? If we try to move the simplified construction we will find that, as o is moved, l and P will drift apart. However, as we have seen in the previous section, we do have a technique for forcing an incidence between a line and a point. What we must therefore do is, after each movement of o, apply our technique to move l and P back together.

We must first find a new determining set that contains both the object o and the point P and does not contain the line l. We can now use the algorithm from the previous section to restore the incidence lP should the two objects drift apart as o is moved.

Chapter 6

Display System Implementation

In this chapter we will discuss the implementation of the display system. This will include a discussion about the applicability of object-oriented design to the domain of planar projective geometry, as well as an introduction to the *Unidraw* C++ library used in the implementation. Finally, we will discuss the development of the display system, both in terms of its use of Unidraw and other algorithms and classes that are independant of Unidraw.

6.1 Object Orientation and the Projective Plane

In this section we will briefly discuss the object-oriented design used in the application developed for this thesis. Class hierarchies are illustrated using the object modeling technique described in [11]. A more detailed discussion of the operation and implementation of the classes will be given later.

One of the problems inherent when discussing an object-oriented design is the difference between *classes* and *objects*. A class describes a set of member variables and methods and the interface for accessing these components. However a class must be instantiated into an object before any of its members can be accessed. (This does not include members declared as static, which can be accessed from the class and do not require an object to be instantiated.) In this section I will use the term *class* to describe a class of objects, and the term *object* to describe the use of an instantiation of a certain class. Since the term object will also be used to describe projective objects (i.e. projective lines and points) on the screen, the context will have to be consulted to determine the correct meaning.

A domain that is often used to illustrate object-oriented design techniques is that of the graphical object editor. This is because there is a large number of features shared by all graphical objects manipulated by the editor. These include attributes as well as methods. For example, both squares and circles have a position and a size. They also both have the operations draw and move. There are also features that are not shared by all graphical objects. For example, objects like rectangles and ellipses two size attributes, one vertical and one horizontal, where as squares and circles only have one size attribute. One of the challenges of object-oriented design is to capture in the class hierarchy both the similarities and differences between the objects being manipulated.

The application of object-oriented techniques is even clearer in the domain of planar projective geometry. This is because of the fundamental duality between lines and points in the projective plane.

Recall that this application was developed using the Unidraw class library. As such, it must conform to the structure imposed by Unidraw. Here we give a simplified design and omit the overhead imposed by Unidraw.

6.1.1 OMT Notation

The Object Modeling Technique (OMT) described in [11] is a graphical notation used for illustrating object models. Since we use OMT to represent our object hierarchies, we now give a brief introduction into the OMT notation we use.

Figure 6.1 shows a simple class hierarchy. The boxes represent classes, with the class names appearing in their boxes. Lines between boxes represent either associations,

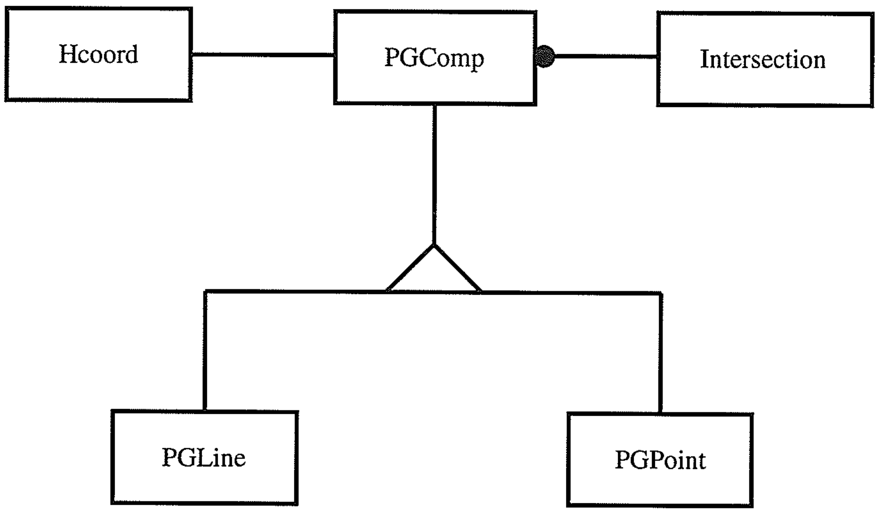

Figure 6.1: Projective Object Class Hierarchy.

aggregations or derived classes. For example, there is an association between *Hcoord* and *PGComp*, and *PGLine* and *PGPoint* are derived from *PGComp*. The latter is indicated by the triangle shown in the connection between the subclasses and their superclass.

Multiplicity of associations is indicated by placing dots on the associative links. For example, in Figure 6.1, each *PGComp* has one *Intersection* associated with it, but each *Intersection* can have one or more *PGComp* objects associated with it.

Aggregations are symbolized by placing hollow diamonds on the lines linking related classes. For example, in Figure 6.3, *BTList* is related to an aggregation of *BTInfo* objects.

6.1.2 Projective Object Hierarchy

Figure 6.1 shows the object model for the projective object hierarchy. Here we express the duality of projective objects in the parent class *PGComp*. This class keeps track of the position of the object (through an association with the *Hcoord* class) and the other objects it is incident with (through an association with the *Intersection* class). It is concerned with the analytic and synthetic properties of projective objects.

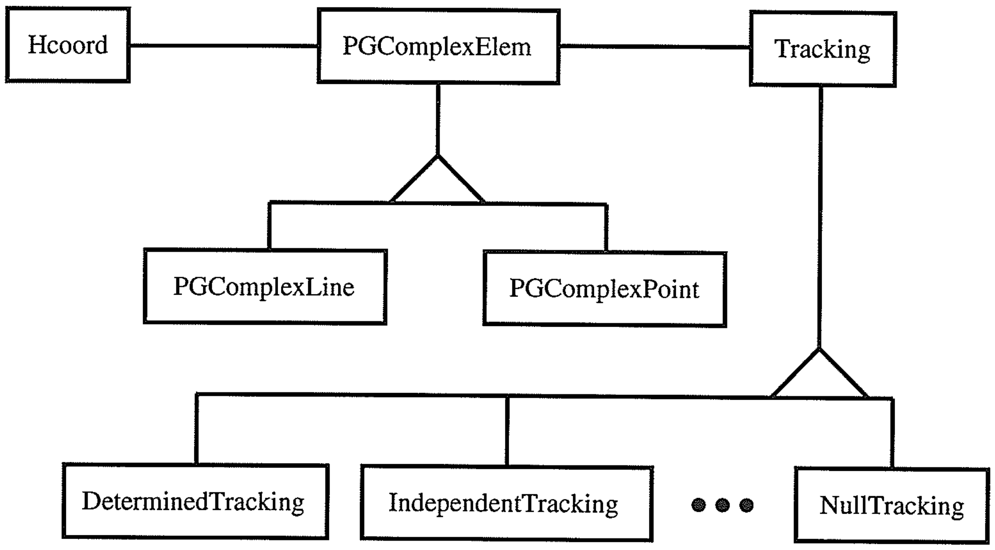

Figure 6.2: Object Movement Class Hierarchy.

The *Hcoord* class is used to store homogeneous coordinates and contains methods for doing dot and cross products. The *Intersection* class stores a list of the other objects an object is incident with. It contains methods for traversing the list and testing membership in the list. This is the only place where the incidence structure of a construction is maintained.

The *PGComp* class has two subclasses, *PGLine* and *PGPoint*. These are concerned with the physical manifestations of the projective objects. Thus they include methods to draw projective lines and points respectively, either on the screen or on a printed page.

6.1.3 Object Movement Hierarchy

We now discuss the class hierarchy involved in the movement of a construction. The operation of moving an object in a construction has been distilled into two associated sub-operations: animation and tracking. Animation moves the object seen by the user from its original position to a new position, and tracking calculates the object's new position. These sub-operations are represented by the associated class hierarchies *PGComplexElem* and *Tracking* respectively. See Figure 6.2. The PGComplexElem class hierarchy is concerned with managing the current position and animation of the object. The position is stored, of course, in a Hcoord object. Since the method of animation is different for points and lines, it is managed by subclasses of PGComplexElem, *PGComplexLine* and *PGComplexPoint*. The animation is performed using Unidraw's Rubberband hierarchy, discussed in Section 6.2.

The Tracking class hierarchy is concerned with the way in which the next position of an object is obtained. The type of tracking to apply depends on whether determining set movement or collineation-based movement is being used, and the specific type of determination or collineation used. Subclasses include *DeterminedTracking* for objects whose position is determined by the intersection of two other objects, *Constrained1Tracking* and *Constrained2Tracking* for objects that are constrained by one other object, and *NullTracking* for objects in the determining set which don't have their positions defined in terms of other objects. The *ShadowTracking* subclass is used when the position of an object is the same as that of another object. This is used when different views of the same object are animated in different windows. There are also several collineation-based Tracking subclasses used in collineation-based movement (discussed in Section 4.2).

The most important method in the PGComplexElem class is the *Track* method. Calling this method on a PGComplex object will cause the object to be animated to its new position, which is obtained from its associated Tracking object. When moving a construction, a list will be created containing one PGComplexElem object for each projective object in the construction (a *PGComplexList*). The entire construction will be animated by simply traversing the list and calling the Track method on each PGComplexElem object in the list.

6.1.4 Determining Set Hierarchy

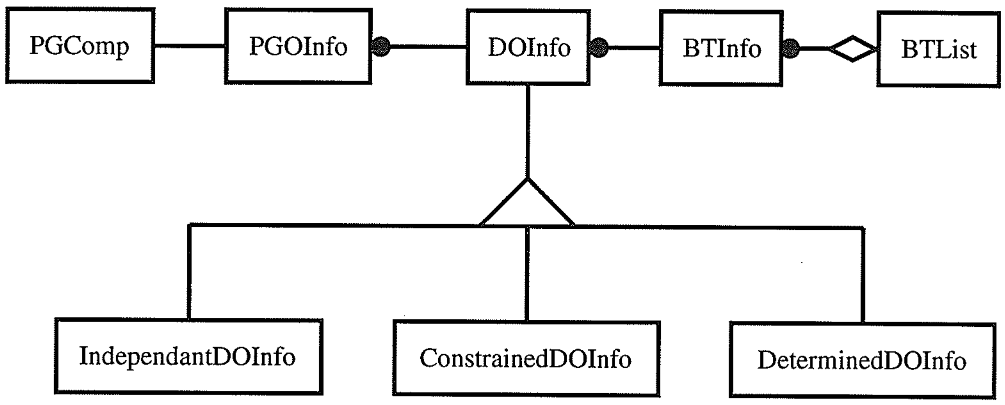

Figure 6.3: Determining Set Class Hierarchy.

The algorithm to find determining sets needs a significant amount of information for each object in the construction. Since this information is somewhat cumbersome and only used in the determining set algorithm, it was placed in a hierarchy separate from the projective object hierarchy. See Figure 6.3.

This hierarchy is based on the *PGOInfo* class, which tracks information about each projective object needed by the determining set algorithm. There is an association to the *PGComp* class, so a PGOInfo object knows about the PGComp object it represents, and vice versa. This class also tracks various status information needed by the determining set algorithm.

One of these pieces of information is an association to the *DOInfo* class. When an object becomes determined, this class is used to define which mechanism was used to determine it. If an object is either constrained or determined, this class contains references to the PGOInfo object(s) that were used to constrain or determine it. This class also contains the method *BuildPGComplexElem* that creates an instantiation of the *PGComplexElem* class based on the method of determination of this projective object.

The DOInfo class is a virtual class used as a superclass for three other classes: *IndependantDOInfo*, *ConstrainedDOInfo*, and *DeterminedDOInfo*. Each of these subclasses corresponds to to one method of determination. IndependantDOInfo indicates that its projective object is in the determining set. DeterminedDOInfo indicates that its projective object is determined, either directly or indirectly, by the determining set. ConstrainedDOInfo indicates that its projective object is constrained on an object that is either independant or determined. Calling the BuildPGComplexElem method on a DOInfo object will call the corresponding method in one of these three subclasses.

The algorithm that finds determining sets (see Section 3.4) operates by adding objects to a working set until it is discovered that the working set is a valid determining set. If a conflict occurs the algorithm performs a *backtrack*. This involves removing the offending object from the working set and *undetermining* all objects that it has determined. The *BTInfo* class is concerned with this backtracking. It stores information that is needed to perform one backtrack. This includes an object in the working set, and a list of objects determined as a result. Creating an instantiation of this class is equivalent to adding an object to the determining set. It contains methods to determine other objects and to backtrack itself.

The BTInfo objects are stored in a *BTList* object. The BTList object stores BTInfo objects in a stack-like data structure. It therefore contains methods to push and pop BTInfo objects onto and off of the stack. This list can be thought of as the working set. This is because it contains the BTInfo objects, and each BTInfo object represents one object in the working set. Objects are added to the working set by pushing BTInfo objects onto the stack. A backtrack operation is performed by popping a BTInfo object off of the stack and calling its backtrack method.

6.2 The Unidraw Class Library

Unidraw is a C++ class library built on the InterViews structured graphics class library developed at Stanford. It was designed to help create graphical editors in a number of domains, such as drawing systems and logic design tools.

The fundamental features common to all graphical editors, such as menu bar commands, mouse-driven tools and graphical objects, are treated as abstract classes. The programmer derives subclasses from these abstract classes to suit the desired domain and introduces them into the Unidraw system. Unidraw then uses the interface defined in the abstract classes to access the new subclasses. As well, predefined subclasses, suitable for many common applications, are included as part of the class library. For example, a simple drawing editor can be made by using the predefined subclasses and deriving only one new subclass.

This introduction to Unidraw is geared toward its use in this implementation. As such, some of the features of Unidraw not used in this implementation may not be discussed.

6.2.1 Structured Graphic Systems

Many graphics libraries exist that allow the programmer to work with primitive graphical objects such as line segments, rectangles and ellipses, and to perform operations with these simple objects. A higher level of abstraction is reached when the programmer is allowed to combine these simple objects into more complex *structured* objects, and then use the same operations on these new, more complex objects. For example, a programmer developing an architectural design package might draw the drafting symbol for a stove by drawing four circles in a square. To actually draw these five objects every time this symbol is needed is tedious. It would be better to combine these objects into a structured object once. This structured object could be now be manipulated and placed with one command.

InterViews is one such package [14]. All graphical objects are derived from the *Graphic* class. One of these subclasses is the *Picture* class. The *Picture* subclass supports hierarchical composition of graphical objects. All graphical objects, including Picture objects, can be drawn, erased, rotated, scaled and moved. A Picture object treats its *children* as a unified block of objects, operating in unison according to the operations performed on their parent.

6.2.2 Structured Graphics with Unidraw

The objects with which the graphical editor is to work are called *components*. Components are seen as having two distinct sets of properties: those pertaining to the component's *subject*, and those pertaining to the component's *view*. The component subjects, represented in Unidraw by the *Component* abstract class, encapsulate the meaning and information of an object, while the component views, represented by the *ComponentView* abstract class, encapsulate the graphical representation of the object that the user sees [15].

The *Graphic* abstract class is a superclass for the basic graphical objects that any graphical editor can draw on the screen, such as lines and rectangles. It encapsulates several attributes that control the appearance of the graphical objects, such as colour, fill pattern or line style. Subclasses of the Graphic class can use these attributes when drawing themselves. These graphic objects are not components themselves, but components can be based on them.

Each Graphic-based object has a *Transformer* attribute. The objects use their Transformers to perform scaling, translation and rotation transformations on their coordinates. This is the usual way that the shape and location of screen objects is changed. The default response to a Move command, for example, is to execute the **translate** method on the transformer of the object being moved, supplying the amount of movement as the translation to use. In this way, many aspects of the appearance of a graphical object can be modified using the Graphic interface, which is common to all graphical objects. In the case of structured Graphic objects, individual objects concatenate their Transformers with those of their parents'. If their parents are themselves children of a structured graphic then they too concatenate their Transformers with those of their parents'. In this way, each object is sure to perform its transformation with respect to its ancestors' coordinate system.

More abstract classes are derived from the component abstract classes. The *GraphicComp* abstract class, derived from *Component*, is used as a superclass for component subjects whose state is represented by Graphic objects. The *GraphicComps* subclass supports the structured composition of GraphicComp objects. The *GraphicView* abstract class, derived from *ComponentView*, is used as a superclass for component views whose appearance is represented by Graphic objects. The *GraphicViews* subclass supports the structured composition of GraphicView objects. The *PostScriptView* abstract class is used as a superclass for component views whose appearance is defined in encapsulated PostScript (usually for the purposes of printing GraphicComp objects).

The behavior of the program's domain is defined by the components. Specifically, the components' reaction to certain tools and commands. For example, a square object and a circle object would each react differently to a reshape command.

Many Unidraw class hierarchies support symbolic class identification. These hierarchies have the **GetClassId** and **IsA** methods that must be implemented by each class in the hierarchy. The GetClassId method returns a symbolic class identifier (of type *ClassId*) that is unique to its class. The IsA method accepts one argument of type ClassId, and returns a boolean value of true if that class has, or has a superclass which has, the same class identifier as given in the argument. Thus, given an instantiation *o* of *LineComp*, a subclass of GraphicComp, calling the IsA method on *o* and passing it the `LINE_COMP` class identifier will return true. Passing the `COMPONENT` identifier will also return true, because LineComp is a subclass (through GraphicComp) of Component.

6.2.3 The Structure of a Unidraw Program

A program using the Unidraw class library begins by declaring several objects that are needed to support the class library. These objects exist for the entire execution of the program and are accessible from any function or object in the program. The *Creator* object is responsible for creating Components (as read from saved files) and creating Views for each Component in the system. The *Catalog* object provides, in general, persistent name to object mapping for component, tool and command objects. It is usually used simply for storing and retrieving work created by the program for the domain in question (i.e. a drawing editor would store and retrieve drawings). The global *Unidraw* object takes the creator and catalog objects as arguments to its constructor, thereby making them accessible to the entire system.

Once the program has declared these objects, it declares one or more *Editor* objects, introduces them to the global unidraw object, and then executes the *run* method of the global unidraw object. The *run* method in the unidraw object takes care of the event loop associated with most graphical user interfaces. Most of the overall appearance of a graphical editor is defined by its Editors. Each Editor object defines the layout of its window (i.e. the placement of menu bar, tools and work area(s), as well as the panning and zooming mechanisms to use), and dictates the menu options and tools available. See figure 6.4 for the appearance of the editor used in the application developed for this thesis.

Any object in the Editor's window that has a shape and size and can accept mouse events is represented as an *Interactor.* These include menu bars, menus, icons and the work area. Interactors contain information regarding their initial shape and their stretchability. The programmer defines the layout of the Editor by specifying the relative arrangement of the objects. When the Editor is drawn, it traverses the hierarchy of interactors, and, using the shape information of each interactor, calculates its size and position. If the Editor's window is resized, the sizes of the interactors in the window are automatically resized.

The program's components are stored in a GraphicComps object that belongs to each Editor object. Each Editor object owns one or more *Viewer* objects A Viewer is the work area which displays the GraphicViews object corresponding to the Editor's GraphicComps object.

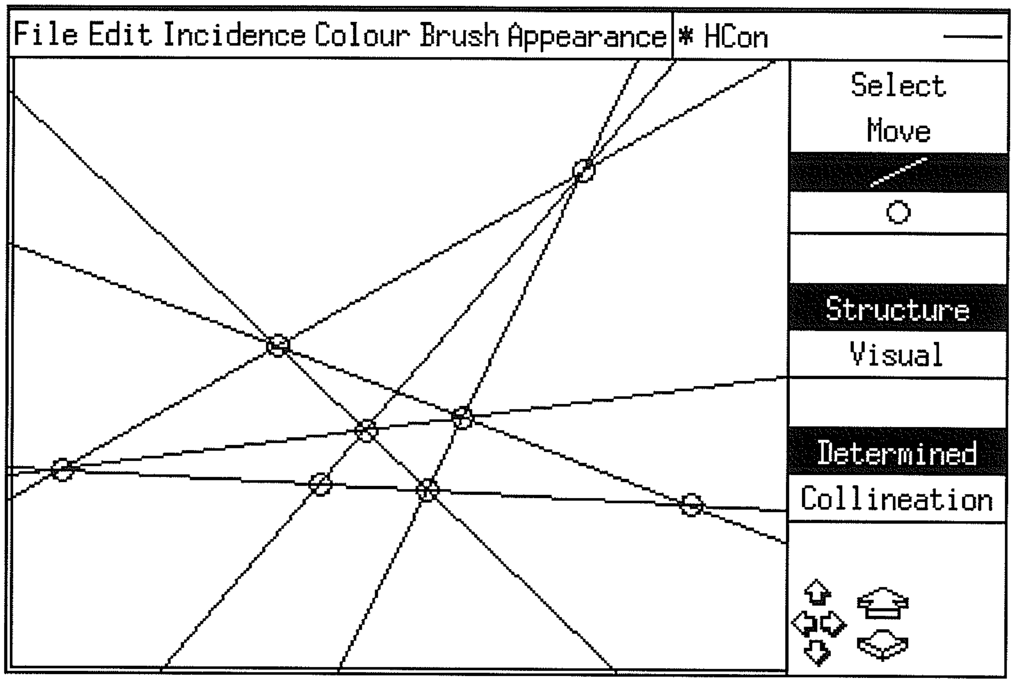

Figure 6.4: The Appearance of our Editor.

6.2.4 Tools and Menus

There are two ways to get user input into a graphical editor. The first method requires the user to first select a tool from a palette of tools. Now, whenever the user clicks a mouse button in the work area, the tool will be activated. This method is appropriate for operations such as placing new objects on the screen or changing the appearance of existing objects. In short, any operation that requires visual feedback (eg. click and drag) is performed by wielding a tool. The other method requires the user to select objects to be operated on beforehand, and then selecting the operation to be performed from a pulldown menu. The operation may open a popup window to solicit more information from the user. This method is appropriate for operations that don't require much (or any) visual feedback, such as storing or retrieving work.

In Unidraw, any operation that can be performed by the user (i.e. through tools or menus) which can be executed, undone or redone is represented by a subclass of the *Command* class. The difference between the Commands for tool-based operations, such as *MoveCmd*, and the Commands for menu-based operations, such as *DeleteCmd*, is that the tool-based operations require parameters. These parameters are supplied by the user wielding the tool.

Unidraw has special classes for using both tool and menu commands. The *Tool* abstract class is a superclass for all mouse-based operations. These include, among others, the *MoveTool* class, used to move objects in the work area, and the *NewCompTool* class, used to place new objects in the work area. The *PanelControl* class is used for presenting a tool icon to the user. It has facilities for displaying text or a graphic symbol in a button and animating the *push down* action when the button is pressed. It also has facilities for grouping several PanelControls together (i.e. a tool palette) so that only one is selected at any time. When a PanelControl object is created, it can be given a *Tool* object that represents the action that will eventually be performed while its icon is selected.

When a Viewer, which is the work area of an Editor, detects a mouse button click, it retrieves the current Tool from its Editor. This Tool is then asked to create a *Manipulator* object that will animate the three phases of the manipulation: *grasp*, *wield* and *effect*. If the action of the Tool that is currently being used depends on the GraphicViews that were selected when the grasp was performed, it may delegate the Manipulator creation to those GraphicViews.

Once the Manipulator has grasped the objects selected in the Viewer it enters its wield phase. Here it animates the selected objects as per the movements of the mouse. When the user releases the mouse button the Manipulator executes its effect phase where it 'cleans up' after the animation. The end result of the manipulation is stored for use when the Manipulator is interpreted.

The Tool is then asked to interpret the manipulation just completed. This can be done by either the Tool itself or by the GraphicViews that created the Manipulator. This involves interrogating the Manipulator to determine the result of the manipulation, and using this information to generate a Command object. When the Command is executed it will make the effect of the manipulation evident in the Editor's components.

The operations required for the animation of a manipulation are performed by a *RubberBand* object that is created by the Tool or GraphicView that creates the Manipulator. Thus the Manipulator is only responsible for the structure of the manipulation, the actual drawing and erasing of objects is performed by the RubberBand.

The RubberBand class hierarchy is used to perform all object animation in Unidraw. The base class defines methods that erase and move an object. Animation is done in XOR mode, so erasing an object is performed by redrawing it. The animation is performed by erasing an object at its old position, finding its new position and drawing it at its new position. Thus to animate an object, all that is usually required is a subclass in this hierarchy that redefines the **Draw** method to draw the object.

As an example, I will now describe the actions that take place when a *MoveTool* is used with a line segment. With the MoveTool selected in the tool palette, the user clicks the mouse button on a line segment in the Viewer's display area. This causes the MoveTool's CreateManipulator operation to be called. It determines which of two possible situations existed when the button was pressed:

- The cursor was on a GraphicView that was one of several GraphicViews that were selected. The MoveTool calls the GraphicView CreateManipulator operation which creates a DragManipulator whose RubberBand is a rectangle that precisely encloses all selected GraphicViews.

- The cursor was on a GraphicView that was not selected. In this case, the selection is cleared and set to the GraphicView that the cursor was pointing to. Since the selected GraphicView is a LineView, the MoveTool now calls the LineView CreateManipulator operation, which creates a DragManipulator whose RubberBand is, in this case, a RubberLine object which has the same size and position as the LineView.

The Manipulator is returned to the Viewer, which invokes the Manipulators grasp, wield and effect phases. The Viewer then calls the Tool's InterpretManipulator operation, passing it the Manipulator. The MoveTool calls the GraphicView's InterpretManipulator operation which uses the coordinates stored in the Manipulator to generate a MoveCmd. The MoveCmd is then executed. This causes the movement stored in the MoveCmd to be applied to the GraphicComps which belong to the selected GraphicViews. The GraphicViews are then redrawn and the user sees the effect of his actions.

Menu commands are implemented by using the *CommandControl* class. This class is used to hold the information needed for one menu option, which includes the name of the command (that appears in the menu when the user clicks on the menu bar) and a Command object that is to be executed when the menu option is selected. CommandControl objects are collected in *PullDownMenu* objects, which are in turn collected in the menu bar.

Commands and Tools need a way to keep track of components or views that the user is especially interested in. This service is provided by *Selection* and *Clipboard* objects.

A Selection object holds a list of GraphicViews. Usually each Editor will have a Selection object for each of its Viewers. This Selection object will then be accessed by the tools and menu commands as required. For example, the *SelectTool* is used to change the GraphicViews in an Editor's Selection object. If the user wants to apply a MoveTool to more than one screen object, those screen objects are placed in the Selection using the SelectTool, and then the MoveTool is wielded. The movement specified using the MoveTool is applied to the GraphicComps belonging to all the GraphicViews in the Selection.

A Clipboard object holds a list of GraphicComps. The GraphicComps that a Command is to be applied to is specified in a Clipboard. The Clipboard may be a default

Clipboard associated with the Command's Editor, or it may be passed to the Command's constructor. For a Command generated by a Tool as part of its InterpretManipulator operation, the Clipboard is compiled by that Tool. For Commands that are invoked from a menu, either the default Clipboard is used or a Clipboard is generated using the GraphicViews in the Editor's Selection. This is what happens when a user selects several objects with a SelectTool and then invokes the DeleteCmd from a menu.

Both the PanelControl class and the CommandControl class have the ability to use *hotkeys*. These are keystrokes that can be used as short cuts for either selecting a tool or invoking a menu command. The keystrokes required for each tool or menu command are specified as parameters to the PanelControl or CommandControl constructors and are automatically displayed in the button or menu line.

6.3 Display System Development

There is an important difference in the operations the two methods of object movement described in this chapter. Determining sets are used to find the movements needed to maintain the incidence structure of a construction. Thus *some* of the objects in the plane are moved. With collineations, we find a transformation of the entire plane, or *all* of the objects in the plane, that produces the desired effect.

The purpose of this thesis was to develop a computer application that would allow a user to create and edit drawings in the projective plane. Now that we have discussed some basic planar projective geometry and have a basic knowledge of the class library used to implement this application, we can discuss the actual development of this application. We will first discuss classes that were developed for the application. While some of these are subclasses of Unidraw classes and were developed to extend Unidraw for the domain of planar projective geometry, others deal solely with planar projective geometry. Some of the latter classes were introduced in Section 6.1 where we discussed their place in the object oriented design of this application. Here we discuss their use in more detail.

6.3.1 New Subclasses

We will now discuss those classes that were extended from the Unidraw hierarchy. It should be noted that, while the Unidraw class library does a good job of abstracting graphical editors, it makes some assumptions about graphical editors that are not true in our case. Therefore, some subclasses were developed solely to counteract these assumptions, and don't really add much functionality to the application. These subclasses will only briefly be mentioned. Also, many of the subclasses do not significantly change the interface inherited from their superclasses. Thus the new interfaces will not be given for these subclasses.

The first new subclass is *PGGraphic*, which is a subclass of *Graphic*. It adds the ability to store a homogeneous coordinate, represented by an Hcoord object, to the Graphic class. This class is used as a superclass for the *PGLine* and *PGPoint* classes, which represent projective lines and projective points, respectively. These subclasses will eventually be used to draw the actual projective objects that the user manipulates in the work area.

PGComp is a subclass of the *GraphicComp* class and encapsulates the behavior common to projective lines and projective points. This behavior includes dealing with homogeneous coordinates and incidence information. The most important extensions are in the **Interpret** and **Uninterpret** methods. These methods deal with commands, specifically the *PGMoveCmd* and the *PGLocusCmd*. These commands are discussed in Section 6.3.3.

Another area where the GraphicComp class needs to be extended is that of external representations. Since we are now tracking homogeneous coordinates and incidence information, this information must be read from and written to save files. Thus the default **Read** and **Write** methods are overridden to store this information. Note that these methods call the respective methods in the GraphicComp class, so that the original behavior is maintained.

Figure 6.5 shows some of the new methods added in the PGComp class. The **GetHcoord** method returns the position of the projective object as a homogeneous coordinate

```
class PGComp {
    Hcoord *GetHcoord();

    void AddIntersection(PGComp *);
    void RemoveIntersection(PGComp *);
    boolean Intersects(PGComp *);
    boolean IsSibling(PGComp *);

    PGComplexList *GetPGComplexList(IntCoord, IntCoord,
        Viewer *);
};
```

Figure 6.5: PGComp extension interface.

triple. Since the GraphicComp class (and therefore the PGComp class) has its information stored in a Graphic, this method simply returns the Hcoord of its PGGraphic. The next four methods deal with the incidence information stored with each PGComp object in its associated Intersection object. **AddIntersection** and **RemoveIntersection** add and remove other objects from an PGComp's Intersection object. **Intersects** returns true if its argument is incident with its PGComp object, false otherwise. **IsSibling** returns true if its argument is incident on any object which is incident with its PGComp object, false otherwise.

Finally, calling **GetPGComplexList** on an object computes a PGComplexList that will animate the construction, with that object being grasped by the user. The arguments are the viewer coordinates where the mouse button was pressed and the viewer the mouse was in at the time. The algorithm used to compute the PGComplexList will be discussed later in this chapter.

The PGComp class is a superclass for *PGLineComp* and *PGPointComp*. These classes add little functionality to the hierarchy. Their main purpose is simply to distinguish lines from points. This is done with the **GetClassId** method.

PGView is a subclass of the *GraphicView* class and encapsulates the behavior common to projective lines and projective points as they appear in the application's work area. The most important extensions are in the **CreateManipulator** and **InterpretManipulator** methods. These methods deal with tools, specifically the *PGCompTool* and the *PGMoveTool*, discussed in Section 6.3.3. The primary reason for overriding the default actions of these methods in the GraphicView class is because projective objects can have complicated incidence structures associated with them, requiring the use of a PGComplexList instead of a simple Rubberband. Another reason is that the positions of projective objects are stored in homogeneous coordinates, not the cartesian coordinates of Unidraw.

Another important point of extension is in the **Update** method. This method is intended to update a ComponentView when the view's Component is changed. In the planar projective geometry domain, the main extension is to update a PGView's homogeneous coordinates when the coordinates of its PGComp change. Also, the user has several different options on how he or she wishes to view the work area (see Section 7). These options affect how a PGView is updated from its PGComp. For example, if the Visual view mode is selected, the PGView's colour is determined from that of its PG-Comp. If Structure view mode is selected, and Determined movement is selected, then the PGView's colour is determined from the mobility (fixed or floating) of its PGComp.

PGView is a superclass for the *PGLineView* and *PGPointView* classes. As with the the PGComp hierarchy, all of the behavior common to both projective lines and points is performed in the parent class, while PGLineView and PGPointView exist primarily to distinguish between lines and points. This is again performed by the GetClassId method.

Since the PGView hierarchy is responsible for the user's view of the projective objects, these classes also ensure that projective lines appear as lines and projective points appear as points. This is performed in three different methods. First, the **CreateHandles** method (inherited from the GraphicView class) is called to create a Rubberband that, when displayed, will highlight a selected view. Second, the **PrintView** class is called to generate a PostScript representation of a projective object. Third, the **CopyPGGraphic** method is called (indirectly) by the PGView::Update method, and returns a copy of a PGGraphic object, which is either a PGLine or a PGPoint.

The *PGComps* class is a subclass of *GraphicComps.* It was derived to extend the Read and Write methods. This was done to facilitate special processing needed when reading and writing incidence information from the save file.

The *PGViewer* class is a subclass of *Viewer.* It was derived for two purposes. First, the default Unidraw method for zooming and panning assumes a finite work area. The projective plane, however, is infinite. Thus, the zooming and panning behavior was modified to implement the operation mentioned in 2.12.2. Second, since the visualization of a homogeneous coordinate triple depends on the current view of the projective plane, the PGViewer class implements the methods **GetPoint** and **GetLine** that return the (cartesian) coordinates of points and lines in the work area, given their homogeneous coordinates.

There are five classes added to the Rubberband hierarchy. *PGLineSelect* and *PG-PointSelect* are the Rubberbands created by the CreateHandles methods described above to implement highlighting. With the default Unidraw objects, this Rubberband is a series of 'handles' around the periphery of the objects. With the points and infinitely long lines of the projective plane, this model doesn't work very well. Instead, we implemented highlighting by drawing noticeably thick borders around highlighted objects.

Note that classes used for highlighting, even though they are derived from the Rubberband class, aren't used for animating. The *PGRubberLine* and *PGRubberPoint* classes implement the animation of projective lines and points. There are three new methods introduced with these two classes. **HardDraw** draws the object without checking or updating the Rubberband's internal **_drawn** flag. This is used to erase objects at the beginning of an animation. (The Rubberband Erase method works in the same way, but it sets the internal _drawn flag to false afterwards.)

Animated PGRubber objects can work in *locus* mode where, in the process of animation, the object leave a trail of its past positions. This is useful for observing the conic sections induced by certain projectivities. The **SetLocus** and *GetLocus* methods provide access to a PGRubber object's internal locus flag.

The *PGComplexRubber* class is used to control the animation of several objects at once using *PGComplexElem* objects discussed later in this chapter. It is passed a PG-ComplexList object which is a list of PGComplexElem objects. The Draw, Erase and Track methods of a PGComplexRubber simply iterate through this list and call the corresponding method on each PGComplexElem.

6.3.2 New Classes

The *Hcoord* class is used to represent homogeneous coordinates. It has methods for accessing the three components of a homogeneous coordinate and performing dot and cross product operations, as well as methods for interfacing to the two-dimensional realm of Unidraw. The public interface to the Hcoord class is given in Figure 6.6.

An Hcoord object can be initialized to zero or to a specific homogeneous coordinate. It is also possible to initialize an Hcoord to represent a given line or point in a given Viewer. The **SetCoords** and **GetCoords** methods can also be used to access the three homogeneous coordinate components of an existing Hcoord object.

Because we are working in a two-dimensional windowing environment, we occasionally need to convert from three-dimensional homogeneous coordinates to two-dimensional screen coordinates. This is performed using the visualization model described in Section 2.12.2. Since this model depends on the position of a viewing plane, each Hcoord ob-

```
class Hcoord {
    Hcoord();
    Hcoord(double, double, double);
    Hcoord(Hcoord *);
    Hcoord(IntCoord, IntCoord, Viewer *);
    Hcoord(IntCoord, Viewer *);
    void SetCoords(double, double, double);
    void GetCoords(double &, double &, double &);
    void CondSetViewer(Viewer *);
    Viewer *GetViewer();
    double DotProduct(Hcoord &);
    Hcoord CrossProduct(Hcoord &);
    Hcoord ScalarMult(double);
    Hcoord VectorAdd(Hcoord &);
    Hcoord Normalize();
    boolean IsEqual(Hcoord &);
    void Random();
    boolean GetLine(double &, double &, double &, double &);
    boolean GetPoint(double &, double &);
};
```

Figure 6.6: Hcoord public interface.

ject that will be displayed contains a pointer to the Viewer object that it will be displayed on. This Viewer object will contain information and methods to implement the visualization model. This pointer can be accessed by using the Hcoord's **CondSetViewer** and **GetViewer** methods. (Note that having this pointer stored in every Hcoord object is not an optimal solution; however, it is necessary due to limitations imposed by Unidraw.)

The **DotProduct** and **CrossProduct** methods implement the dot product and cross product operations from linear algebra. **ScalarMult** and **VectorAdd** will perform scalar multiplication and vector addition, respectively. The **Normalize** method multiplies the three floating point components by a scalar so that the vector has length one. This is done because operations like CrossProduct can cause the homogeneous coordinate components to increase exponentially, and eventually cause overflows. Thus, applying Normalize on a regular basis helps to minimize this problem.

The **IsEqual** method returns true if its object and argument are the same homogeneous coordinate. If the two triples are not exactly equal, this involves trying to show that one is a scalar multiple of the other. The **Random** method redefines its object to be a random homogeneous coordinate.

The **GetLine** and **GetPoint** methods return the screen coordinates of this Hcoord as a line and a point respectively. These are primarily convenience functions, since they call their respective methods in the Hcoord's Viewer.

In Chapter 5 we represented the change in a homogeneous coordinate as a linear function of three variables, x, y and z. In the implementation this concept is represented by the *Delta* class. This class represents the change in a homogeneous coordinate triple by three linear terms, one for each of the three floating point values of the coordinate. The linear terms are represented by objects of the *Dcoord* class. In the Dcoord class, a linear term of the variables x, y and z is given as a triple of floating point values, where each value is the coefficient of one of the variables.

For example, suppose we had $\delta = (r_1x + s_1y + t_1z, r_2x + s_2y + t_2z, r_3x + s_3y + t_3z)$.

```
class Delta {
    Delta();
    Delta(Dcoord &, Dcoord &, Dcoord &);
    void GetCoords(Dcoord &, Dcoord &, Dcoord &);
    Delta CrossProduct(Hcoord &);
    Dcoord DotProduct(Hcoord &);
    Delta ScalarMult(double);
    Delta VectorAdd(Delta &);
};
```

Figure 6.7: Delta public interface.

This would be represented as a Delta object $D(d_1, d_2, d_3)$. The three Dcoord components would be $d_1(r_1, s_1, t_1)$, $d_2(r_2, s_2, t_2)$ and $d_3(r_3, s_3, t_3)$.

The public interface for the Delta class is shown in Figure 6.7. (The Dcoord interface isn't shown because it is essentially equivalent to the Hcoord class). This class also contains methods for performing linear algebra operations, also as in the Hcoord class.

The incidence information for a PGComp is maintained by an *Intersection* object associated with that PGComp object. The Intersection object is simply a list of other PGComp objects incident with that PGComp object. This list is derived from the Unidraw UList class. Its interface is shown in Figure 6.8

The **Append**, **Remove** and **IsMember** methods are self-explanatory. For iterating through the list of incident objects, the Intersection class uses the Unidraw *Iterator* interface. To start at the beginning of the list, call the **First** method with the Iterator to use. To advance to the next object in the list, call the **Next** method. The **Done** method tests if an Iterator has passed the end of the list. Finally, the **GetAdj** method returns the PGComp object that the Iterator is currently pointing to.

```
class Intersection : public UList {
public:
    Intersection();
    virtual ~Intersection();

    void Append(PGComp *);
    void Remove(PGComp *);
    boolean IsMember(PGComp *);
    void First(Iterator &);
    void Next(Iterator &);
    boolean Done(Iterator );
    PGComp *GetAdj(Iterator );
};
```

Figure 6.8: Intersection public interface.

```
class HTransformer {
    HTransformer();
    Hcoord Transform(Hcoord *);
    Hcoord InvTransform(Hcoord *);
    void AdjustPhi(double);
    void AdjustTheta(double);
    void AdjustScale(double);
    double GetScale();
};
```

Figure 6.9: The HTransformer public interface.

The visualization model used in this implementation, as mentioned in Section 2.12.2, involves a viewing plane that can be rotated around the origin to view different parts of the drawing (i.e. panning). In fact, in this implementation the viewing plane is not rotated, but instead the three-dimensional Euclidean lines and planes that represent the drawing are rotated. This rotation is performed by three-dimensional matrix multiplication on the homogeneous coordinates of the objects being drawn. As well, the size of the drawing on the viewing plane can be modified by a scaling factor. These operations are performed by the *HTransformer* class. The public interface to the HTransformer class is shown in Figure 6.9.

Each PGViewer has a HTransformer object, which is used to transform homogeneous coordinates before being displayed in the PGViewer, and to inversely transform homogeneous coordinates that are obtained from the mouse in the PGViewer. Each HTransformer object is initialized to the identity transformation. That is, it will not transform or scale the homogeneous coordinates of objects being drawn. To introduce a transformation, one of the **Adjust** methods can be called. The **AdjustPhi** and **AdjustTheta** methods cause a rotation around the x and y axes, respectively. The **AdjustScale** method multiplies the scaling factor, initially one, by its argument. Calling the **Transform** method with an Hcoord object multiplies the homogeneous coordinate by the transformation matrix and multiplies the third component of the homogeneous coordinate by the scaling factor. Calling the **InvTransform** method with an Hcoord object performs the reverse procedure on that Hcoord.

Moving an object in a projective drawing is more complicated than with a more familiar drawing application, because the incidence structure must be maintained at all times. As a result, grasping and moving one object may necessitate the simultaneous movement of several objects. While this problem was discussed in Chapters 4 and 5, here we discuss the classes used in solving this problem. These are the *PGComplexElem* class hierarchy and the *Tracking* class hierarchy, introduced in Section 6.1.3. The interfaces are given in Figure 6.10 and Figure 6.11.

The PGComplexElem class is an abstract class, and as such cannot be instantiated. It is instead used as a superclass for other classes, *PGComplexPoint* and *PGComplexLine.* Objects instantiated from these subclasses will perform the actual animation of lines and points.

These subclasses can be instantiated in two ways, as is indicated by the existence of two constructors in the parent class. The first way is for objects which are grasped by the user. There will only be one of these at any one time. In this case, the object can be initialized with an optional Tracking object. This indicates that the users movements must be constrained with another object, whose position will be given by the optional Tracking object. The second way to instantiate a PGComplexElem subclass is when it's position does not *directly* depend on the mouse movement. In this case, a Tracking object *must* be supplied to the constructor. Whenever the object is to move, the Tracking object will be consulted to find the new position of the object.

It should be mentioned that most of the methods of this class are pure virtual functions which are not implemented in the abstract class. When we speak of the behavior of the

```
class PGComplexElem {
public:
    virtual ClassId GetClassId() = 0;
    virtual void Draw() = 0;
    virtual void HardDraw() = 0;
    virtual void Erase() = 0;
    virtual void Track(IntCoord x, IntCoord y) = 0;

    virtual PGComplexElem * CreateShadow(Viewer *v,
        boolean duality) = 0;
    virtual void SetComp(PGComp *c);
    virtual PGComp *GetComp();
    virtual Hcoord *GetHcoord();
    void SetLocus(boolean = false);
    void SetDelta(Delta *);
    Delta *GetDelta();
    void ComputeDelta();
protected:
    PGComplexElem(Hcoord *,Viewer *, Tracking * = nil);
    PGComplexElem(Viewer *, Hcoord *, Tracking *);
};
```

Figure 6.10: The PGComplexElem interface.

```
class Tracking {
public:
    virtual ~Tracking();
    virtual void TrackHcoord();
    virtual void ComputeDelta();
    void SetDelta(Delta *);
    Delta *GetDelta();
    Hcoord *GetHcoord();

protected:
    Tracking(Hcoord *);
};
```

Figure 6.11: The Tracking interface.

PGComplexElem class or objects instantiated from it, we are in fact speaking of the behavior of its subclasses.

The **Draw** and **Erase** methods respectively draw and erase the object. The **HardDraw** method will draw the object even if it is already drawn. This was necessary because Unidraw doesn't erase objects when it starts an animation, so calling HardDraw at the beginning of an animation effectively erases it. Since PGComplexElem objects use Rubberbands for the animation, these drawing methods simply call the corresponding methods on the Rubberband object that is controlled by the PGComplexElem. The **Track** method updates the position of the object to reflect movement of the mouse and/or constraining objects. This method is passed the current position of the mouse pointer. This position is only used if this object is grasped by the user. Otherwise, the Tracking object is consulted to find the next position of the object.

So far we have only discussed animating objects on the same Viewer in which they are being moved. Occasionally we want to animate the movement of an object on more than one Viewer. These *shadow* objects mimic the movement of their *subjects* on other Viewers. The **CreateShadow** method creates a shadow of the object on which it was called. The parameters include the Viewer on which the shadow is to be animated, and a boolean flag indicating whether or not a dual shadow is to be created.

When moving an object on the screen, the Unidraw system only keeps track of the object that was grasped. If other objects move as a result of the tool being wielded, we must keep track of those other objects. Therefore, each PGComplexElem has an internal pointer which points to the Component it is animating. The **SetComp** and **GetComp** methods provide access to this pointer. The **GetHcoord** method returns the object's current position. This is used when the movement is finished and the PGMoveCmd is constructed to enforce the movement on the constructions components. The **SetLocus** method calls SetLocus on the PGComplexElem's PGRubber object.

The algorithm for forcing incidences in Section 5.2 involves using a determining set in much the same way as construction movement with determining sets. That is, assigning a new position to one object and finding the effects of position in the rest of the construction. When forcing incidences, however, we are working with the change in position as opposed to the position. This change in position is represented with objects of the Delta class, mentioned above.

Because the algorithm's use of determining sets is similar to that of construction movement, Delta information is included in the PGComplexElem and Tracking hierarchy. This includes the methods **SetDelta**, **GetDelta** and **ComputeDelta**. Thus, when PGComplexElem objects are grouped into a PGComplexList, the algorithm can find the resulting change in position of every object in the construction by simply traversing this list and calling the ComputeDelta method on each object. Note that these three methods are actually implemented only in the Tracking hierarchy; the methods in the PGComplexElem class are convenience functions that perform operations on a PGComplexElem's Tracking object, if one has been defined.

The Tracking class is also an abstract class, with a subclass for each type of tracking that can be performed. The **TrackHcoord** method is called to compute the next position of an object. This position is then retrieved by calling the **GetHcoord** method. The change in position of the object is computed by calling the **ComputeDelta** method and retrieved by calling the **GetDelta** method. Note that the ComputeDelta method only has to be implemented for subclasses whose operation is based on determining sets.

Objects of the *PGComplexList* class are used to group all PGComplexElem's required for a movement. This class contains methods for maintaining and iterating through the list of PGComplexElems. It also has a method **ComputeDelta** that iterates through the list, calling ComputeDelta on each PGComplexElem.

We now discuss classes used in the algorithm that finds determining sets. These classes were first introduced in Section 6.1.4. The implementation of the algorithm that uses these classes is discussed in Section 6.3.4. Recall that the algorithm to find determining sets operates by adding objects to a working set until it is discovered that the working set is a valid determining set.

The first class we discuss is the *PGOInfo* class. See Figure 6.12. This class stores information about projective objects that is only of use to the determining set algorithm. This information includes the determined status and the incidence information of a projective object. Note that the incidence information of projective objects (as stored in PGComp objects) is duplicated in the PGOInfo objects. The construction stored in the PGComp objects will be duplicated in PGOInfo objects, but in a form more suitable for the determining set algorithm.

The constructor creates an instance that corresponds to the given PGComp. Once a PGOInfo has been created for every projective object in the construction, the **InitAdjList** method is called on every PGOInfo. This method initializes the incidence information in the PGOInfo. Since the incidence information is in terms of PGOInfo objects, this operation cannot be performed in the constructor, when not all projective objects may have been assigned PGOInfo objects. Thus this operation is performed in a separate method, after all PGOInfo objects have been created.

The **GetObject** method returns the projective object (a PGComp) that corresponds to the PGOInfo object. The **GetHcoord** method is a convenience function that calls GetHcoord on this projective object.

The **GetDeterminedChildren** method searches the objects incident with this PGOInfo object for PGOInfo objects that are determined. Only the first three such objects are returned, since if there are three or more such objects, there is an error in the working set. The **CountDeterminedChildren** method performs the same search but only returns the number of such objects found. The **Degree** method returns the number of objects incident with this PGOInfo object. Finally, the **RemoveIncidence** method removes an incidence from the incidence information stored with the PGOInfo.

The incidence and determining information is accessed through the public data members **_alist** and **_doinfo** respectively. While these data members should perhaps be private and have public methods available to access them, this was not done to simplify the code using this class. Since the use of this class is so restricted, this lack of data hiding was deemed to be permissible.

As well, the class also contains the **_mask** data member. This member is used to store special requirements for objects in the determining set. For example, in the incidence forcing algorithm, we need to find a determining set which contains a point P but not a line l. Thus the PGOInfo object for l can have its _mask set to `RS_NoIndependant`. This will ensure that l will not be in any determining set found.

Next we discuss the *DOInfo* class, also shown in Figure 6.12. This class, as was mentioned in Section 6.1.4, is used as a superclass for the classes which indicate the way in which a PGOInfo object is determined, if it is determined. The three subclasses used are IndependantDOInfo, ConstrainedDOInfo and DeterminedDOInfo. When a PGOInfo object is first created, its _doinfo data member is set to `nil`. When it becomes determined, _doinfo is set to point to an instance of one of these three subclasses. The test that is performed to see if a PGOInfo is determined simply tests if this member is not `nil`.

When creating an instance of a DOInfo subclass, the DOInfo constructor, called by its subclass constructors, requires three pieces of information. These are the PGOInfo which is being determined, the rank at which the PGOInfo is determined, and any objects used to determine the PGOInfo. The **GetStatus** method is used to find out how an object is determined. It returns a value of the enumerated type *ObjectStatus*. This enumerated type contains only three values: `ObStat_independant`, `ObStat_constrained` and `ObStat_determined`.

The **BackTrack** method *undetermines* the object determined by this DOInfo. This simply involves setting the associated PGOInfo's _doinfo pointer to `nil`. Finally, the **BuildPGComplexElem** method creates an instance of PGComplexElem that corresponds to this determined object.

```
class PGOInfo {
public:
    PGOInfo(PGComp *o);
    void InitAdjList();
    PGComp *GetObject();
    Hcoord *GetHcoord();
    void GetDeterminedChildren(PGOInfo *&,PGOInfo *&,
        PGOInfo *&);
    int CountDeterminedChildren();
    int Degree();
    void RemoveIncidence(PGOInfo *i);

    PGOInfo **_alist;
    DOInfo *_doinfo;
    ReserveStatus _mask;
};

class DOInfo {
public:
    DOInfo(PGOInfo *, int = 0, PGOInfo *= nil, PGOInfo *= nil);
    virtual DOInfo *Copy() = 0;
    virtual ObjectStatus GetStatus() = 0;
    void BackTrack();
    virtual PGComplexElem *BuildPGComplexElem(Viewer *v) = 0;
};
```

Figure 6.12: PGOInfo and DOInfo public interfaces.

```
class BTInfo {
public:
    BTInfo();
    ~BTInfo();
    void First(Iterator &i);
    void Last(Iterator &i);
    void Next(Iterator &i);
    void Prev(Iterator &i);
    boolean Done(Iterator i);
    DOInfo *GetInfo(Iterator i);
    BTInfo *Copy();
    PGOInfo *BackTrack();
    void FixObject(PGOInfo *object);
    void ConstrainObject(PGOInfo *object, PGOInfo *constraining);
    void DetermineObject(PGOInfo *object, PGOInfo *ref1,
        PGOInfo *ref2);
};
```

Figure 6.13: BTInfo public interface.

We now discuss the *BTInfo* class, shown in Figure 6.13. An object of this class, as we know from Section 6.1.4, represents one projective object in the working set, and a list of projective objects that can be immediately determined from it. To add an object to the working set, we create a BTInfo object and call either its **FixObject** method or its **ConstrainObject** method. Recall that a determining set can be either augmented or unaugmented, depending on whether the objects contained in it are completely independant or if some are constrained. The FixObject method is used to add an independant object to the working set, and the ConstrainObject method is used to add a constrained object to the working set. In both cases, the object to be added to the working set is passed as a parameter. In the latter case, the constraining object is also passed. These methods not only store the working set object in the BTInfo, but also assign it a DOInfo object.

This class has six iterator-based methods for maintaining a list of DOInfo objects. Objects are determined by calling the BTInfo's **DetermineObject** method and passing it the object to be determined and the determining objects. This method assigns the object a DeterminedDOInfo object, and adds that DeterminedDOInfo object to the list.

This information is needed, among other things, for the **BackTrack** method. This method calls BackTrack on each DOInfo object in the list of DOInfo objects and then calls BackTrack on the DOInfo object representing the object in the working set. This method returns the working set object it represented.

The final class we look at in this section is the *BTList* class. This class stores BTInfo objects in a stack-like data structure to support the backtracking algorithm. Usually, only one BTList object exists at a time. Once a BTInfo object has been created (indicating an object in the working set) it is *pushed* onto the BTList. To perform a backtrack, a BTInfo is *popped* of the BTList and its BackTrack method is executed. Recall that the BTInfo BackTrack method returns the PGOInfo corresponding to its working set object. The algorithm was iterating through the undetermined objects when it selected this object to add to the working set. Since the BackTrack method returns this object we know where

```
class BTList : public UList {
public:
    BTList(BTInfo *i = nil);
    virtual ~BTList();
    BTInfo *Pop();
    void Push(BTInfo *i);
    BTList *Copy();
};
```

Figure 6.14: BTList public interface.

to resume the iteration after a backtrack. Thus we know we will select a different object than the one which caused the backtrack.

6.3.3 New Tool and Command Subclasses

We will now discuss the subclasses that represent new tools and commands. The *PGCompTool* is used to place new PGComp object in the PGViewer. It was derived from the *NewCompTool* class because the default action of the tool included clearing the the Viewer's Selection and setting it to the grasped object. The PGCompTool allows the user to select other objects before placing the new object, and thus allow the rapid creation of constructions as described in Chapter 7.

The *PGMoveTool* was derived from the *MoveTool* class because its default action is move all selected objects as a block, where as we need to grasp only one object. The *PGMoveCmd* was derived from *MoveCmd* because we are representing positions as homogeneous coordinates instead of cartesian coordinates.

The following Command subclasses are used to implement the new pull-down menu commands discussed in Chapter 7. The *IntersectCmd* is used to implement the *Inter-*

section menu command. It requires that two or more objects of the same type (either lines or points). The *PGMobilityCmd* changes the mobility attribute (fixed or floating) of PGComp objects. The *PGPrintCmd* prints the construction in the work area. This was used instead of the default Unidraw *PrintCmd* to deal with the different way we use the work area (i.e. the infinite canvas). The *PGDualCmd* opens another editor containing the dual construction. I.e. the roles of lines and points are reversed. The *PGStyleCmd* implements the *Set Line Endpoints* menu command. The *PGForceCmd* is used to force incidences. It requires that the user has selected one line and one point that are not as yet incident (at least structurally if not in the visible work area). It then implements the incidence forcing algorithm of Section 5.2. The *PGSeparateCmd* is used to remove an incidence between two objects in a construction. It requires that the user has selected a line and a point which have an incidence between them in the incidence information maintained by the program. This command then removes this incidence so that, for example, the line and point can now move off of each other.

The *PGLocusCmd* changes the locus command is used to turn the locus effect on or off for selected objects. The *PGCollineationCmd* is used to define collineations. In the implementation of collineation-based movement, the user can define two central collineations, known as the *red* and *green* collineations. The central collineations are defined by selecting a point (the centre) and a line (the axis). These collineations are used to determine the movement of the construction when an object is moved. This command operates as a toggle. Thus, any selected object that is part of a collineation is removed from that collineation, and vice versa.

6.3.4 Implementation of Algorithms

We now discuss the implementation of the algorithms introduced earlier in this thesis. These include the algorithm to find determining sets and the algorithm to force incidences.

Determining Set Algorithm

The determining set algorithm requires the objects of a projective construction to be in a form which can be iterated. Thus the first step is to create an array of PGOInfo objects which contains the same incidence information as the projective construction. We do this by performing a breadth-first traversal of the projective construction. For each object visited we create a PGOInfo object. The PGOInfo objects are added to the array in the order their PGComps are visited. After the array is completed, the incidence information is copied from the PGComp objects. As well, PGComp objects that are fixed (indicating they are to be in the determining set) have their PGOInfo objects determined with an IndependentDOInfo object.

Next the array is checked to make sure that the fixed objects do not induce any conflicts, thereby making it impossible to find a determining set.

We now implement the algorithm of Section 3.4. The implementation uses a BTInfo object as the stack Q, and the array of PGOInfo objects as the object set Σ. The algorithm consists of two steps in a loop. The first step adds an object to the working set. The object added is chosen by calling the function **PickFix**. This function iterates through the array (possibly from a given starting point) and returns the first undetermined object it finds that is not immediately adjacent to any other determined objects.

If no such object is found, the function returns `nil`. In this case, the algorithm calls the function **PickConstrain** to try to select a constrained object to add to the working set. This function performs a search similar to that of the PickFix function, except that it looks for undetermined objects that are adjacent to exactly one determined object. If no such object is found, the function returns `nil`.

If no object can be found to add to the working set, a backtrack is performed. We again attempt to find a new working set object. We loop until either a new working set object is found, or until the stack (BTList) is exhausted. In the latter case, there is no determining set and the algorithms exits.

The second step in the loop creates a BTInfo object for the new working set object. Then the **DetermineObjects** function is called. This function will find all undetermined objects that can now be determined, and appends them to the list maintained by the BTInfo object. The function uses a nested loop. The inner loop iterates through the object array, calling **GetDeterminedChildren** on each PGOInfo in the array. If an object has two determined children, it is determined using the BTInfo's DetermineObject method. If it has three determined children, there is a conflict in the working set and the function returns with an error condition. The algorithm now knows that it must perform a backtrack and try again. The outer loop continues until the inner loop fails to find any new determined objects, after which it returns normally.

If the array still contains undetermined objects we look for another working set object. Otherwise we have found a valid determining set. In this case, the algorithm orders the PGOInfo array on the rank of determination, and returns this array.

Recall that the BackTrack method on a BTInfo object returns its working set object. Thus, when searching for a working set object after a backtrack, we know where to start the search.

Forced Incidence Algorithm

Section 5.2 includes an algorithm for forcing an incidence between two projective objects, a point P and a line l. To do this, we need a determining set containing P but not l. This is done as in the previous section.

The determining set is then converted to a PGComplexList. Recall that PGComplex-Elem objects contain Delta objects which are used in the incidence forcing algorithms. This conversion is performed by running through the ordered array returned by the determining set algorithm, and for each PGOInfo object, calling the BuildPGComplexElem method on its DOInfo object. The resulting PGComplexElem objects are added to the PGComplexList. Finally, the Delta object belonging to P's PGComplexElem is set to

(x, y, z). Note that the default value for a Delta object is $(0,0,0)$.

Recall that the algorithm consists of a loop which runs as long as $P \cdot l$ is not close to zero. The first step in the loop is to compute the Delta's of all objects in the construction. This is done by calling the ComputeDelta method on the PGComplexList.

The next step in the loop is to find the constraining equation. This equation is of the form $l \cdot P + l \cdot \delta_P + \delta_l \cdot P = rx + sy + tz + c = 0$. Recall from our discussion of the Dcoord and Delta classes that the Delta class has a DotProduct method that operates on an Hcoord object. This method is used to calculate $l \cdot \delta_P + \delta_l \cdot P = rx + sy + tz$. Note that this method returns a Dcoord object, since the result is a linear term in x, y and z. The constant term c is computed using the Hcoord DotProduct method.

Next we solve the constraining equation. This is done using the last technique of Section 5.2.1. The result is a floating point triple (x, y, z) which we add to the homogeneous coordinates of P. We now recompute the Delta's of the PGComplexList. This operation has the side effect of recalculating the positions of the constructions objects. The loop then tests $l \cdot P$, and if it is not close enough to zero, we repeat the loop. Otherwise, we apply the movement to the construction and denote P as being incident with l.

Note that there is a safety counter in the loop test to guard against pathological cases that cause the algorithm to fall into a near-infinite loop. After the loop has iterated a fixed number of times (30), the algorithm exits. The movement found for P is applied, but the incidence information is not updated.

Algorithms for Collineations

Collineations are used to implement another form of object movement. See Section 4.2. Once the user has defined two central collineations, any object in the construction, which is not part of a collineation, can be moved by the user. The collineations will then be consulted to determine the movement of the rest of the construction.

Note that it is possible to find matrices that represent collineations. Multiplying the homogeneous coordinates of objects in the plane by such a matrix will perform the transformation of the collineation on the plane. However, when using collineations for movement as we do, this would require that the matrix be recalculated and re-applied for each step in the movement. Therefore, the more efficient (but somewhat inelegant) method described in this section is used instead.

Consider the case where only one collineation is defined, say a homology with centre V and axis l. See Figure 6.15. In this case the movement of the construction is quite restricted. Suppose we grasp a point P and wish to move it. Any valid (V, l) collineation will move P along a fixed line m that passes through V and P. If we find the line containing V and the original position of P, any (V, l) collineation will move P along this line.

When creating PGComplexElem objects to perform a movement, we know that we can constrain the grasped object to always be incident with another object of the opposite type. I.e. we can specify a line that the grasped point P will always be incident with. No matter where the user tries to move the point, it will always *slide* along the constraining line. Thus we can use this technique to implement moving a single object under one collineation.

Consider the case where we have another point S. See Figure 6.15. When the user grasps and moves P, we are actually finding the collineation that performs the movement that the user desires. Since this collineation affects every object in the plane, it will affect S. To find this effect, we notice two key incidences that must be maintained by the collineation. First, just as with P, there is a fixed line n through V and S with which S must always be incident. Second, the line through P and S intersects l at a fixed point T. The line PS must always pass through T, so when P is moved, S must move accordingly. The position of S can be found by calculating $(P \times T) \times n$ for any given P.

With these two constraints we can find the effect of the collineation on any point in

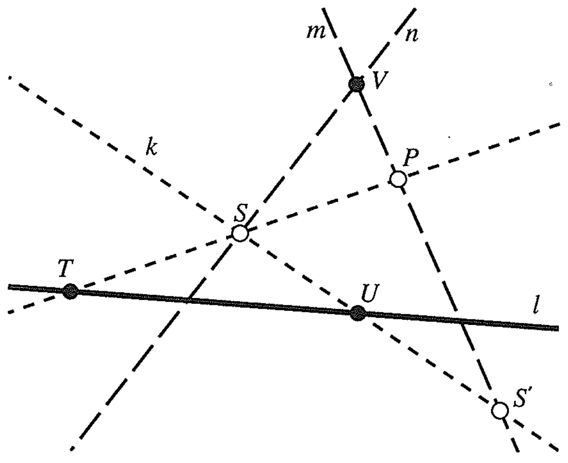

Figure 6.15: Finding the effects of a Collineation.

the plane that is not P and not on l (because they are fixed). Also, the point cannot be on m. This is because the line PS will not move if S is on m and P moves along m.

Suppose that we now add a line k to our construction, such that k passes through S. Notice that k intersects l in a (fixed) point U. Using the technique described above, we can find the effect on S of any collineation that moves P. Since S and U both lie on k, we can find the position of k by calculating $((P \times T) \times n) \times U$ for any given P.

If we are given a line k and we want to know how a collineation will affect it, we can *reverse-engineer* this process. That is, given a line k, find U and choose a random point S on k. Then we find the effect of P on S. As we have just seen, this can be used to find the effect of P on k.

At this point, we can find the effect of a collineation on a point, say S, which is not on the line m (S), and the effect of a collineation on a line, say k. We now describe a method for finding the effect of the collineation on a point which is on the line m. Let S' be this point. The approach we use is similar to the one used above. We find an arbitrary line k that passes through S' and use the above technique to find the effect of moving P on k. Since S' is at the intersection of k and m, its position is easily found by calculating $(((P \times T) \times n) \times U) \times m$ for any given P.

We can use the above constructions to implement transformation by a single collineation. To do this, we derive *Tracking* subclasses for each of the three possible cases. These objects do not actually draw the *intermediate* objects that are used in the construction, but instead just work with their homogeneous coordinates.

When these objects are created, they initialize themselves with the information they need to track the movement of any other point or line based on the movement of the grasped point. For example, an object for the first and simplest case finds the homogeneous coordinates of the line PS and then of T. It also finds the homogeneous coordinates of the line n. It stores T, n and a pointer to P internally. Whenever the **TrackHcoord** method is called on that object, the expression $(T \times P) \times n$ is evaluated to find the new position of S for any position of P.

Notice that we have so far only considered the case where the user grasps a point. If the user grasps a line, we simply reverse the roles of lines and points in the constructions, and rely on the principle of duality to ensure success.

Notice also that we have so far only worked with one collineation. Using two collineations we are not as restricted in the movement of the grasped point. Suppose we have the centres and axes of two collineations, (V, l) and (V', l'). See Figure 6.16. If the user grasps the point P and moves it to P', we can find a point P'' that is at the intersection of lines VP and $V'P'$. This is a point on VP where a (V, l) collineation could have moved P. We can now find a (V', l') collineation that moves P'' to P'. Thus we can find two collineations which, when applied in sequence, move P to P'. We now apply these same collineations to the rest of the plane.

There is an important difference in the operations of the methods of object movement described in this chapter. Determining sets are used to find the movements needed to maintain the incidence structure of a construction. Thus *some* of the objects in the plane are moved. With collineations, we find a transformation of the entire plane, or *all* of the objects in the plane, that produces the desired effect.

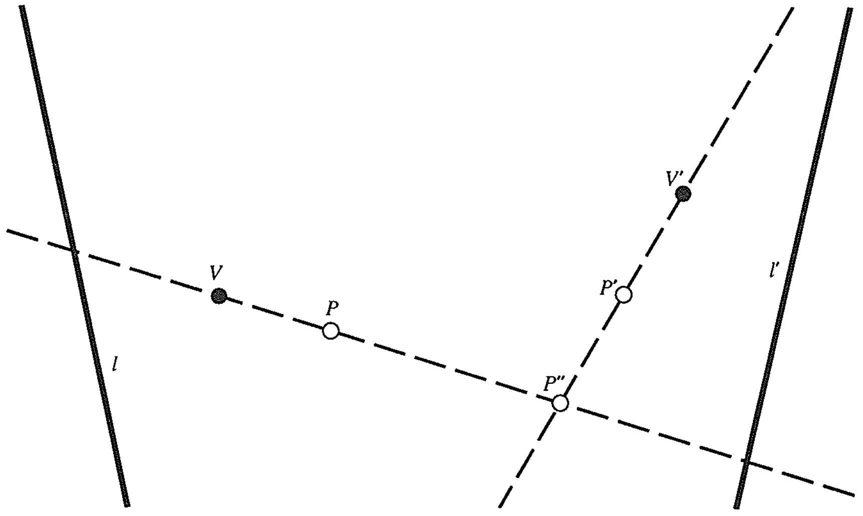

Figure 6.16: Moving a point with two Collineations.

Object Movement Algorithms

We can now unify the previous sections to describe what happens when the user activates a PGMoveTool on an object in the work area. The PGView CreateManipulator method calls GetPGComplexList on the the PGComp that the tool was clicked on. This method decides whether determining set movement or collineation-based movement is in effect, and calls either GetDetPGComplexList or GetColPGComplexList respectively. A PG-ComplexRubber object is created with this PGComplexList and the PGComplexRubber is used to create a DragManipulator. This manipulator is returned from the method.

We now discuss the way PGComplexLists are created. The GetDetPGComplexList function first determines if the construction is a (3,3)-configuration. If this is not the case, then the construction has a determining set. We call the determining set algorithm and build a PGComplexList from the determining set.

If this is a (3,3)-configuration, the construction has no determining set, and we must use the technique of Section 5.3 to move the construction. Notice that we only deal with (3,3)-configurations here. This is because they are the simplest constructions that have no determining set, and therefore the simplest constructions with which we can test our technique. While it may be possible to extend the technique for more complex configurations, here it is specialized for the case of (3,3)-configurations.

The first step is to convert the construction to an array of PGOInfo objects, just as before. We then select an incidence to cut. To select the incidence, we first look for an object that is not fixed or adjacent to a fixed object. Then we select an incidence of that object that is as close to the end of the array as possible. This incidence is cut in the PGOInfo array.

We then find a determining set for the simplified construction and build a PGComplexList of the determining set. To this list we add the information about the incidence that was cut. This information will be used by the Track method of the PGComplexRubber to call the incidence forcing algorithm for each step of the movement.

The GetColPGComplexList function finds a PGComplexList that will move the construction using collineation-based movement. It first finds the objects that make up the centres and axes of the collineations. Then it creates a PGComplexElem for the grasped object. It then loops through all (non-fixed) objects in the construction. For each object it creates a Tracking object and a PGComplexElem object. Recall that there are three different Tracking subclasses used with collineation-based movement. The decision of which one to instantiate for an object is based on its relationship to the grasped object. For example, if the grasped object is a point and we are creating a Tracking object for a line, we create an instance of the second collineation Tracking subclass.

The PGComplexElem objects are combined into a PGComplexList, which is returned.

6.3.5 Algorithm Complexity

We will now say a few words on the complexities of the algorithms discussed in the last section. These algorithms all have their complexities based on n, the number of objects in the construction. The determining set algorithm, since it is a backtracking algorithm, is exponential in n. The incidence forcing algorithm iterates until the desired incidence is achieved, up to a maximum iteration count. Each iteration involves one iteration through the PGComplexList, performing a cross product for each element in the list. Each cross product operation requires nine floating point operations. Thus each iteration requires $9n$ operations and is $O(n)$. Since the iteration limit of the algorithm is 30, it requires at most $30 \cdot 9n$ operations, and is therefore $O(n)$.

The object movement algorithm (once the PGComplexList is found) runs as long as the user wields the PGMoveTool. The complexity of each step in the movement depends on whether determining set or collineation-based movement is used, and whether or not the construction is a (3,3)-configuration. For the simplest case, the construction is not a (3,3)-configuration and determining set movement is used. Each step in the movement simply involves iterating through the PGComplexList and computing the positions of objects using the cross product operation. Thus, as above, this step requires $9n$ operations and is $O(n)$.

If the construction is a (3,3)-configuration, then after iterating through the PGComplexList, the incidence forcing algorithm is called. Thus every step in the movement now requires several runs through the PGComplexList. The number of times the list is iterated is still bounded by a constant, so the algorithm is still $O(n)$.

For collineation-based movement, each object requires at most five cross products to be computed, or 45 floating point operations. Thus each movement step requires $45n$ steps and is $O(n)$.

Chapter 7

User Guide

We will now discuss how a user uses the application to create and modify constructions in the projective plane. The main screen layout for the program is shown in Figure 6.4. There is a menu/status bar at the top of the window, a series of control panels on the right side, and a panning/zooming control in the lower right corner. The status bar contains status indicators for the file name of the current construction, its modifications status, and a colour/brush indicator showing the colour and brush that new objects will be drawn with. The large central area is the main drawing area. This is where the construction currently being edited is visible, and where the user manipulates objects in the construction. We now discuss the layout in detail.

7.1 Menus

The program has six menus along the top of the window. They are the *File*, *Edit*, *Special, Colour, Brush*, and *Points* menus. These menus contain commands that are executed by activating the desired menu with the mouse and then selecting the proper command from that menu.

The following commands are available under the *File* menu. *New* clears the current construction from memory and allows the user to create a new construction. *Open* allows the user to load in a construction that was saved to a file. *Save* allows the user to save a construction as a file. *Print* allows the user to generate a PostScript representation of the contents of the drawing area, *as it currently appears.* This representation can either be saved as a file or printed on a PostScript printer. *Quit* exits the program. *Display Dual* opens another editor that displays the dual of the construction currently being manipulated. The editor implements the dual correlation described in Section 2.10. This dual editor can be used to create new lines and points as well as to move objects. When both editors are visible at the same time, any change in one is immediately seen in its dual. Note that some of the menu commands are not available in the dual editor window.

The following commands are available under the *Edit* menu. *Undo* reverses the effect of the last command. *Redo* reverses the effect of the last Undo command, essentially redoing an undone command. The program maintains a list of the last 20 commands executed, allowing the user to undo the last 20 operations performed. These include menu commands as well as operations performed with tools (see below). Using the Undo and Redo commands, the user can move backwards and forwards through this list. However, as soon as a new command is executed, all subsequent commands are deleted. *Delete* deletes the objects in the selection.

The *Incidence* menu contains commands that are unique to the domain of planar projective geometry, specifically those that affect incidence structure. *Intersection* creates points at the intersections pairs of lines and creates lines that connect pairs of points. The user first selects a set of two or more lines *or* points (not both), and executes this command. For example, if n lines are selected, then this command will create points at the intersections of the $\binom{n}{2}$ pairs of lines that are selected. This commands also updates the incidence information to indicate that each point created is incident with both of the lines defining it. Note that any point that is already at the intersection of two of the selected lines will not be created again by this command.

Fix Mobility and *Float Mobility* are used to alter the mobility of objects. An object's mobility can be either *Fixed* or *Floating* (the default). Fixing an object's mobility ensures that it will be in all determining sets computed for its construction. The effects of this are discussed in Section 4.1.

In Section 4.2 we show how collineation-based movement can be used once the user has selected the centres and axes of two central collineations. The two central collineations are known as the *red* collineation and the *green* collineation. To specify a point and a line to be the centre and axis of the red collineation, the user selects the point and line and executes the *Red Collineation* command. These objects then appear in red. Similarly, to specify the centre and axis of the green collineation, the user executes the *Green Collineation* command. If only one collineation is defined, the movement performed will be the more restricted, one-collineation movement.

Force Incidence is used to move a point onto a line when both are part of the same construction. This commands uses the algorithm of Section 5.2. The point and the line to be forced must be selected before this command is executed. *Disconnect* removes an incidence between a selected line and a selected point. The objects do not move apart, but the incidence between them is removed from the incidence structure of the construction. If part of the construction is moved, the incidence will no longer be maintained.

The *Colour* and *Brush* menus are both used to affect the cosmetic appearance of objects in the construction. The *Colour* menu is used to either change the colour of selected objects or to change the colour that all new objects will be created with. The *Brush* menu is used to either change the dash pattern and width of selected lines or of all new lines that will be created.

The *Appearance* menu is used to change the appearance of the objects in the construction. *Set Line Endpoints* is used to affect the appearance of lines. Occasionally, especially in complicated and cluttered constructions, it is not necessary to see every line in its entirety. This command can be used to select two points on such lines that will act as endpoints. That is, only the portion of the line *between* its endpoints will be visible. Note that the ideas of *between* and line segments are not valid in the projective plane; this operation is only meant to make constructions easier to comprehend or more aesthetically pleasing. As such, we use the Euclidean definition of *between* on the Euclidean representation of the projective construction that appears in the drawing window. A line is drawn as a segment only when both its endpoints are in the drawing window.

The *Filled* and *Unfilled* commands allow points to appear as either filled circles or hollow circles respectively. This menu is also used to turn *locus effects* on or off for objects currently selected. If the locus effect is on for an object, it will leave a trace during animation when part of the construction is moved.

7.2 Control Panels

Along the right side of the window are three control panels. They are the *Tool*, *View* and *Movement* control panels. Each has a number of choices, only one of which is active at any time. The user can modify the behavior of the program by altering the choice selected in one of the panels. This is done by activating the desired choice with the mouse.

7.2.1 The *Tool* Palette

The *Tool* panel controls which tool is applied when the left mouse button is pressed in the main drawing area. The *Select* tool allows the user to add objects to the program's *selection*. The program's selection is used by the user to indicate which objects he or she would like to work with. For example, some of the commands perform a specific action to the objects in the selection. Thus the user uses the selection to specify the objects the menu command is to work with, and then executes the menu command.

To add an object to the program's selection, move the mouse pointer to the object to select, and press the left mouse button. When objects are selected, their appearance is highlighted. Normally, selecting an object clears the previous selection, so only one object is selected at a time. If the user holds down the *shift* key on the keyboard while selecting an object, the previous selection is not cleared. In this way, the user can have several objects selected at once.

As a shortcut, the right mouse button always implements the select tool. Thus using the right mouse button in the main drawing area is equivalent to selecting the select tool in the control panel and using the left mouse button in the main drawing area.

The *Move* tool is used to move an object on the main drawing area. This is done by positioning the mouse pointer over the object to move, activating the tool (by pressing the left mouse button), moving the object to its new position, and releasing the mouse button. The object is then redrawn in its new position.

If the object being moved is part of construction, the other objects in the construction will also be moved to maintain the incidence structure of the construction. The type of movement to use is specified by the *Movement* control panel.

As a shortcut, the middle mouse button always implements the move tool. Thus using the middle mouse button in the main drawing area is equivalent to selecting the move tool in the control panel and using the left mouse button in the main drawing area.

The line tool is used to place projective lines in the main drawing area. To place a new line in the drawing area, move the mouse pointer to the main drawing area and press the left mouse button. A horizontal line is drawn through the mouse pointer. Moving the mouse while holding down the left mouse button moves the line in the drawing area. Releasing the mouse button places the line in its current position.

Since a line is usually defined by two parameters (i.e. two points on the line or an intercept and a slope), it is difficult to control a line using a single pointing device. Therefore, we have associated mouse movement to line movement in the following way: Moving the mouse in the vertical direction will cause the line to move in the vertical direction. Moving the mouse in the horizontal direction will cause the line to rotate about a point in the centre of the drawing window. This is somewhat analogous to defining the line movement as a change in intercept and a change in slope. Using these semantics the line can be placed anywhere in the drawing area.

The final tool is the point tool. It is used to place points in the drawing area in much the same way that lines are placed using the line tool. Since a point can be controlled with just one pointing device, the movement semantics for a point are much more straight forward.

Note that the movement semantics for lines and points are the same for objects that have just been created and are being placed and for objects that already exist and are being moved with the MoveTool.

Creating Incidences

Since it is often desired to specify incidences between lines and points as a construction is being created, there are some shortcuts that have been implemented to specify incidences as points and lines are placed.

To indicate that the point being created is to be incident with an existing line, select the line before placing the point. When the point is moved on the display area, it will be constrained to the selected line. As well, the incidence information will be updated to indicate that the newly created point is incident on the selected line.

Suppose that it is desired to create both a line on an existing point and a point on the new line. The line and point can be created at the same time. To do this, select the existing point that the line is to be incident with. Then create and place the new point as before. The line connecting the selected and new points will automatically be drawn. The incidence information will then be updated to indicate that the new line is incident with both points.

In fact, when ever one or more points are selected when a point is being placed, connecting lines will be drawn between each of the selected points and the point being placed, and the incidence information will be updated automatically.

For example, to create two points incident on a line, create one of the points using the point tool, select that point, and then create the second point. A line will be drawn between the first point and the one currently being placed. As well, the incidence information will be updated to indicate that the line is incident on the points.

Analogous incidences can be created when placing lines. That is, having a line selected while placing a new line will create a point at the intersection of the new and selected lines. Having a point selected will constrain the new line to be incident with the existing point.

7.2.2 The *View* Control Panel

The next control panel controls the view of the diagram that appears in the drawing area. When the *Structure* view mode is selected, the appearance of the diagram reflects the fixed objects or collineations defined. Note that the appearance depends on the movement type currently in effect. If determining set movement is in effect, fixed lines and points will appear in red. If collineation-based movement is in effect, the centres and axes of the red and green central collineations will be red and green respectively. All other lines and points will appear in black. Any cosmetic attributes set with the menus (such as colour, brush style or line endpoints) is ignored.

When the *Visual* view mode is selected, the cosmetic attributes of objects are used. Since it is possible to create objects that are invisible in the visual mode, it is usually best to perform all manipulation in structure mode, and only use visual mode for viewing or printing.

Note that the view mode is consulted when the *Print* menu command is executed. Thus this program can be used to create aesthetically pleasing planar projective geometry diagrams on paper. In fact, all planar projective geometry drawings in this thesis were produced with this program.

7.2.3 The *Movement* Control Panel

The next control panel controls the mechanism used to move objects in the diagram. The two selections are *Determined*, which uses the determining set method, and *Collineation*, which uses the collineation method. These methods are discussed in Chapter 4.

7.3 Panning and Zooming

In the lower right corner of the window is the panning/zooming control. This allows the user to alter the viewing transformation that is used in the visualization algorithm discussed in Section 2.12.2. By using these controls, the user can move the viewing plane in order to see a different part of the projective plane, or to see a larger portion of the plane, or to see the plane in greater detail. To perform either panning or zooming, click the mouse on the appropriate arrow button.

Another method of panning is to *grab* the plane and move it. This is done by holding down the control key and pressing the middle mouse button while the mouse pointer is in the drawing area. The user can then directly move the plane any way he or she wishes.

Chapter 8

Experience with Program

We will now discuss the performance of the application developed for this thesis.

The algorithm that finds a determining set for a construction tries to systematically add objects to a working set until one of two events occurs: The set is discovered to be a determining set, or the set is shown to contain conflicts. If the former event occurs, the algorithm is finished. If the latter event occurs, the algorithms backtracks to the point where the last object was added, removes it and tries a different object. The backtracks can in fact cascade and cause more backtracks. Thus the algorithm is potentially very expensive.

However, since there are, in general, several determining sets that exist for a given construction, the algorithm usually finds a solution with a minimum of backtracking. For example, the simplified Desargues' construction (the Desargues' configuration with one incidence removed) is one of the most complicated constructions with which the program has been tested. In most of the test movements, a determining set was found with two or fewer backtracks performed, with the rare movement requiring ≈ 20 backtracks. Even in the latter case, the movement was performed with no noticeable delay for the user. It should be noted that, after 100 backtrack operations, the algorithm assumes that no determining set exists and tries to use the algorithm below to move the construction.

The algorithm that was used to force an incidence between a line and a point uses a technique that is not guaranteed to find the best solution. In fact, in some pathological cases, no solution is found, or the solution that is found is a degenerate solution. Degenerate solutions collapse one or more lines and one or more points onto each other. Thus, even though the incidence information maintained by the program is satisfied, all points and lines are no longer distinct and the construction becomes invalid.

This algorithm was applied in several different ways to several constructions. The success ratio of this operation (i.e. the percentage of trials that successfully forced an incidence) was 97.5 percent. The failures occurred because the algorithm was locked into a degenerate state before the incidence could be achieved. Of the successful trials, 14.4 percent found a degenerate solution. The average number of iterations required was 3.86.

Figure 8.1 shows how this algorithm was used to complete a 10_3 configuration [2]. The figure on the left is the construction before the algorithm was called. The algorithm was then asked to force the filled point and the dashed line to be incident. The figure on the right is the construction after the algorithm finished.

In the process of this experimentation, it was discovered that degenerate solutions can be avoided by constraining the determining set chosen by the algorithm (see Chapter 4), or by moving the objects being forced closer together.

The algorithm used to move constructions that have no determining set uses the algorithm used to force incidences. It operates by moving the construction to the best of its abilities using only a simplified determining set, and then calling the incidence forcing algorithm to keep the construction consistent. Thus it is equivalent to multiple invocations of the algorithm discussed above, and is subject to the same liabilities.

It is important to note, however, that each step in the movement is very small, so the amount the incidence-forcing algorithm must do each time it is called is also relatively small.

Note also that, because of the interactive and dynamic nature of this operation, quan-

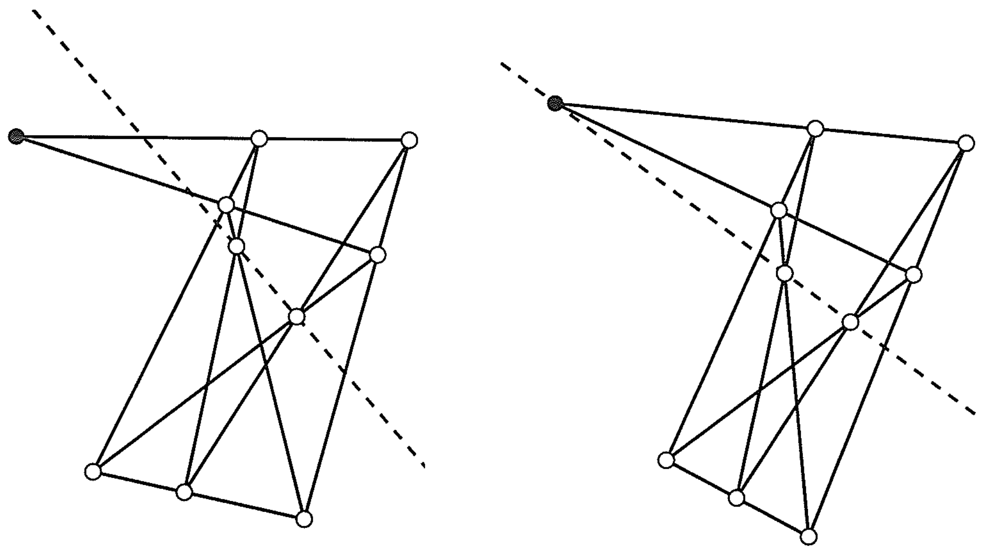

Figure 8.1: Results of applying the incidence forcing algorithm.

titative results are elusive. We can, however, report that most movements can operate over a restricted area without entering a degenerate state. Drastic movements do tend to lead to a degenerate state. As well, once a degenerate state is reached, it is difficult to return to a non-degenerate state without using the applications *Undo* feature. For certain movements, constraining the determining set, as mentioned above, helps to avoid degenerate states.

The operation of panning is useful for viewing points at infinity. For example, in Figure 8.2, we have two views of the harmonic conjugate construction. On the left we see that one of the harmonic conjugate points is off the screen. In fact, it is a point at infinity. By panning the projective plane, we can move this point into the main viewing area. Notice that, on the left, the point is at the intersection of two parallel lines. On the right, the lines are no longer parallel.

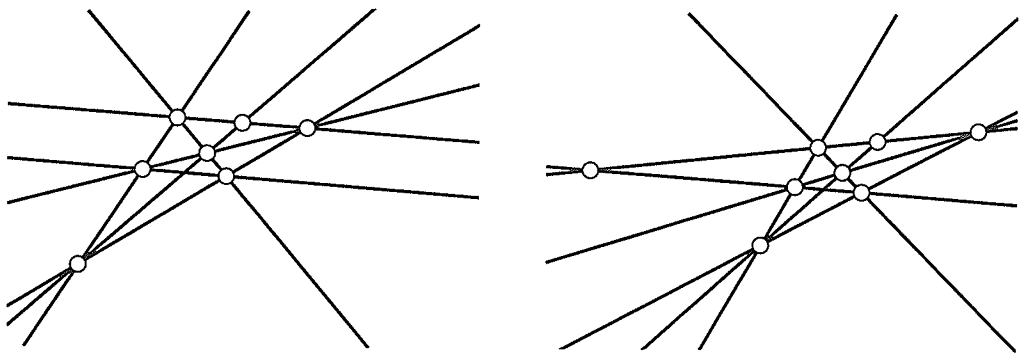

Figure 8.2: Results of Panning across the Harmonic Conjugate construction.

8.1 Further Work

In preparing this thesis and the associated implementation, we found several operations that would be helpful in an application such as this and that could be added to this implementation to improve its usefulness.

There exists a relationship between conic sections and polarities that would be interesting to explore in an animated environment such as this. This relationship relates a pole and its polar through its interactions with a conic section in the projective plane. Thus it would be required to define and display conic sections with the same ease that we can currently work with lines and points. It would also require a *Tracking* class to link the pole and polar through a conic.

As well, all though it is beyond the scope of this work, the ability to work in projective three-space would be most interesting. While this would require completely redesigning the user semantics of this implementation, it would be useful, since many of the applications of projective geometry reduce to the study of projectivities between projective planes in projective three-space [9].

Another opportunity for interesting work lies with the operation of forcing incidences. Specifically, the study of degenerate solutions. Our experience showed that minor changes to the determining set used by the algorithm (applied by fixing objects in the construction) could cause an operation that failed previously to succeed. Thus the study of how the choice of the determining set can affect the performance of the algorithm could prove fruitful.

When degenerate solutions do occur, the result, while different from the original, is a valid construction in the projective plane. Thus a study of *collapsed* constructions, and the different constructions that can be *collapsed* from a given construction, could also prove interesting.

Bibliography

[1] H. Benke, et al., *Fundamentals of Mathematics, Volume 2, Geometry*, MIT Press, 1983.

[2] Jürgen Bokowski, Bernd Sturmfels, *Computational Synthetic Geometry*, Lecture Notes in Mathematics, 1355, Springer-Verlag, 1989.

[3] James Foley, et al., *Computer Graphics: Principle and practice*, Addison-Wesley Publishing, 1990.

[4] David Kelly, personal communication.

[5] Mark A. Linton, John M. Vlissides, Paul R. Calder, *Composing User Interfaces with InterViews* Center for Integrated Systems, Stanford University.

[6] *Curriculum and Evaluation Standards for School Mathematics*, National Council of Teachers of Mathematics, Reston, Virginia, 1989.

[7] Online manual pages, InterViews 3.1 distribution.

[8] Daniel Pedoe, *An Introduction to Projective Geometry*, Pergamon Press, 1963.

[9] Michael A. Penna, Richard R. Patterson, *Projective Geometry and its Applications to Computer Graphics*, Prentice-Hall, New Jersey, 1986.

[10] Franco P. Preparata, Michael Ian Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[11] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice Hall Inc., 1991.

[12] A. Seidenberg, *Lectures in Projective Geometry*, D. Van Nostrand Company, 1962.

[13] Frederick W. Stevenson, *Projective Planes*, W. H. Freeman and Company, 1972.

[14] John M. Vlissides, Mark A. Linton, *Applying Object-Oriented Design to Structured Graphics*, Proceedings of the USENIX C++ Conference, Denver, Colorado, October 1988.

[15] John M. Vlissides, Mark A. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors", *Proceedings of the ACM SIGGRAPH/SIGCHI User Interface Software and Technologies '89 Conference*, 1989.

[16] Olive Whicher, *Projective Geometry: Creative Polarities in Space and Time*, Rudolf Steiner Press, 1971.