Hanna Wallach

Visual Representation of CAD Constraints

Computer Science Tripos, Part II

Newnham College

2001

Contents

1

Introduction

Within the architectural industry, there is wide agreement that the next generation of architectural design tools must provide support for parametric design. Such design tools enable relations between geometric entities to be specified using constraints. Specifying relationships between geometric entities allows a solution family – a set of elements that are topologically equivalent in terms of their constituent parts and the relationships between them – to be defined. The ability to create solution families is extremely powerful: multiple design options can be created very quickly, allowing the solution space to be fully explored. In addition to this, the speed with which design alterations can be made enables computer-aided design to become a far more interactive and creative process. However, the nature of constraint-based design requires users to work at a new level of abstraction. Architectural firms currently address this issue by employing programmers to use constraint-based design tools on behalf of architects. This is clearly a usability problem.

A crucial development would be to eliminate or reduce this usability problem, enabling architects to use constraint-based CAD tools themselves. However, addressing this usability problem is not solely an issue of user interface design. Rather, the challenge is that of designing a programming language to be used by non-programmers. Such programming languages are known as *end-user programming languages*.

The aim of this project is to create a constraint-based CAD package with a constraint representation that will address the usability problems found in existing constraint-based CAD packages. This is clearly an interdisciplinary task, requiring work to be undertaken involving constraint solvers, end-user programming language design, and design of constraint-based CAD systems. A set of usability requirements for the project emerged from structured interviews with users of constraint-based CAD systems at an international architecture and design practice.

Several possible design approaches were considered initially. The one finally adopted is a novel concept, based on methodologies originally proposed for the display of layered information on air traffic control displays.

This report describes the preparation undertaken for the project, as well as the implementation and evaluation processes.

2

Preparation

The chapter summarises the work undertaken prior to implementation. The requirements of my proposed system are outlined, and current research in relevant areas of computer science is discussed. The software engineering techniques employed in the project are outlined, and a description of the tools used in implementation is given.

2.1 Requirements Analysis

Prior to writing my project proposal, I met with the developer of Custom Objects[1], an existing constraint-based CAD package. This resulted in the identification of the core components of my project and the technical requirements of each component.

In order to gain a thorough understanding of the usability requirements of constraint-based CAD systems, I arranged to spend a day at Foster and Partners (an international architecture and design practice). Structured interviews were conducted with four users of existing constraint-based CAD systems. The interviews were based around the following topics:

- The types of constraints regularly encountered in architectural and design work.
- The capabilities and limitations of existing constraint-based and dimension-driven CAD systems.
- The types of visual constraint representation that architects and designers would like to use.
- The reconciliation of the creative aspects of architecture and design with the more mathematical approach of defining constraints.

Discussion of these topics with users enabled me to establish a set of usability requirements for the project.

2.1.1 Requirements

The geometric primitives provided should be sufficient to allow the creation of simple geometric structures. Direct manipulation of geometric entities must be possible. Creation of entities that are not part of the final drawing but are used in the process of constraining other entities (*i.e.* construction entities) should be possible. These entities should be visually distinguishable from entities that are part of the drawing itself.

The constraints provided should include geometric relationships such as coincidence, as well as dimensioned constraints such as the Euclidean distance between points and lines.

The primary requirement of a constraint solver for use in CAD systems is the ability to satisfy constraints sufficiently fast that real-time interaction with the system is possible. In addition, the ability to handle under-specified constraint systems is desirable.

The visual representation should ensure that there is a clear mapping between each constraint and the geometry constrained by it. In addition, the type of each constraint and any values used (if appropriate) should be visible to the user. Constraint creation and editing processes should involve direct manipulation of the constraint representation.

Facilities should be provided for saving drawings and subsequently reloading them. The data representation for files should be clearly structured, easily extensible, and in a format that could be used by software. An obvious choice for such a data representation is XML.

2.2 Research

The aim of this section is to provide the reader with an overview of current research in the areas of computer science relevant to the project. The areas covered are constraint-based CAD, constraint solving techniques, visual language theory and the Extensible Markup Language (XML). Investigation of these research topics took the form of reading relevant research papers, web-pages and books.

2.2.1 Existing Constraint-Based CAD Systems

Existing constraint-based CAD systems were analysed with respect to geometric primitives and constraints provided, constraint solver used, visual representation of constraints and interaction style.

Geometric Primitives

The primary geometric primitives provided by both conventional and constraint-based CAD packages are generally a subset of points, line segments, lines, circles, arcs, polylines, polygons[20]. More complicated primitives, such as Bezier curves, are unnecessary since all drawings must be converted to the primitives above prior to the realisation of the product or building being designed.

In addition to primary geometric primitives, some CAD systems provide *construction objects*[33, 12] (geometric entities that are not part of the final drawing, but are used in the process of constraining entities belonging to the final drawing). Construction objects (usually points, lines and circles) are generally created and manipulated in an identical fashion to their primary counterparts.

Constraints

Some systems[33, 30, 12] provide constraints that act upon entire geometric primitives. Other systems[15] only allow the user to specify constraints on points. Once the constraints have been enforced, determining the locations of the points, the rest of the drawing is filled in. The constraints supported by both these approaches can be divided into two categories:

- Dimensioned constraints, such as constraining the Euclidean distance between points and a lines (or line segments), the angle between two lines (or line segments), or the radius of a circle.
- Non-dimensioned relationships, including congruence, coincidence, tangency and orthogonality.

It is important that the constraints implemented are not dependent on the particular constraint solver used, thus allowing other solvers to be substituted without significantly affecting the system as a whole.

Constraint Solvers

A wide variety of techniques for satisfying constraints have been employed in constraint-based CAD packages. For instance, Juno[15] and Converge[30] use Newton-Raphson iteration, Briar[12] uses differential constraint methods, and ThingLab II[18] uses local propagation. Constraint solving is a substantial research field in its own right, used in many different areas of computer science, and there are many approaches to constraint satisfaction, each of which has advantages and disadvantages with respect to suitability for use in CAD systems.

Visual Representation of Constraints

Existing constraint-based CAD packages tend to use one of three types of constraint representation: textual languages for describing constraints, schematic representations, and graphical annotations directly superimposed on the geometry.

Textual languages, used in systems such as Juno[15], have the advantage of being editable. However, if the usability of such textual languages is analysed using a technique such as *Cognitive Dimensions of Notations*[13], several problems are highlighted:

- The *closeness of mapping* between the programming world and problem world is distant.
- Creation of textual description has a higher *abstraction barrier* than direct manipulation.
- Textual descriptions of constraints tend to be more *diffuse* than graphical representations.
- Describing constraints textually makes a higher demand on *cognitive resources* than direct manipulation.

Schematic representations, such as the dependency graph used in Custom Objects[1], have an advantage over textual representations because they can be directly manipulated. However, usability problems still arise due to the separation of constraint representation and drawing.

Visual representations (superimposed on the drawing) are used in Converge[30] and Briar[12]. These have significant advantages over textual and schematic representations. In particular, integration of the constraint representation and drawing makes dependencies between geometry and constraints explicit. Also, the same gestures can be used for creation and manipulation of both constraints and geometry.

Interaction Style

A major reason for the popularity of direct manipulation drawing programs is the fluidity and ease with which geometry can be created. It is crucial that the creation and manipulation of constraints does not detract from this. Some constraint-based CAD systems[30, 33] allow the user to create and manipulate constraints in exactly the same fashion as that used for geometric entities. This has the advantage that only one interaction paradigm has to be mastered by the user. Additionally, users of a system that makes use of the same method for creation or manipulation of both geometric entities and constraints will be far less likely to make errors due to confusion over which method should be used for creation or manipulation of any particular entity. Other systems[12] make use of more radical approaches. However, such approaches can impair usability.

2.2.2 Constraint Solving Techniques

There are many different approaches to constraint solving. When choosing a constraint solver for a CAD system, the types of constraint supported by different constraint solving techniques must be considered. It is also crucial to consider the efficiency of the constraint solver – the ability to solve constraints at speeds that enable real-time interaction with the system is vital.

Constraint solvers for CAD applications should be able to handle ambiguous or under-constrained systems. A constraint problem is well-constrained if the number of remaining degrees of freedom after the constraint have been specified is zero. In other words, a unique solution to the constraint problem exists. Clearly this is desirable. An over-constrained system is where the number of remaining degrees of freedom is negative – too many constraints are specified for a solution to be found. An under-constrained system is where the system has a positive (and non-zero) number of remaining degrees of freedom, thus resulting in several valid solutions. This situation often arises in constraint-based CAD. Therefore, when choosing a constraint solver for a constraint-based CAD system, the constraint solver should ideally handle such cases without requiring additional input from the user.

Local Propagation Solvers

Local propagation solvers are often used in graphical applications. They operate by propagating degrees of freedom, states or values. Constraints are represented as an undirected graph, with variables as nodes and constraints as edges. The solver then attempts to direct the graph so that the value of each variable can be determined. Local propagation is very efficient and supports arbitrary domains. However, LP solvers cannot simultaneously handle multiple constraints.

Iterative Numeric Solvers

Iterative numeric solvers use methods such as Newton-Raphson, and Sutherland's relaxation technique. They can handle simultaneous non-linear constraints, but are not usually efficient enough to be used as the main constraint solver for CAD systems (their performance can be $O(N^3)$). Also, iterative techniques can only identify the locally optimal solution. In some cases they may not converge on a solution at all, even if one exists. Iterative numeric solvers are often used in situations where other methods have failed.

Direct Numeric Solvers

Direct numeric techniques, such as Gaussian elimination, attempt to determine an exact solution using symbolic manipulation of constraint equations. They are able to solve simultaneous linear equations, but may not be able to handle under-constrained systems. Direct numeric solvers tend to be more efficient than iterative numeric solvers, but less efficient than some local propagation solvers.

Geometric Solvers

Geometric solvers[33] are closely related to local propagation solvers. Geometric degrees of freedom are symbolically analysed, often using a graph to represent the constraints. Despite their similarity to local propagation solvers, geometric solving algorithms can be more complicated, thus making them somewhat more difficult to implement.

Choice of Constraint Solving Technique

The primary requirement of a constraint solving algorithm for a CAD system is speed. It was therefore felt that iterative numeric techniques would be unsuitable for use in this project. The more general constraint solving techniques available allow greater flexibility in the design and implementation of constraints. This was considered to be advantageous. Of the constraint solving techniques considered, local propagation is the most general. Finally, ease of implementation was felt to be important. Local propagation solvers are relatively simple to implement, highly efficient and extremely general.

It was therefore decided that local propagation was the most suitable constraint solving technique for this project.

2.2.3 Selection of a Local Propagation Algorithm

Local propagation solvers can vary in several ways:

- Support for one-way or multi-way constraints.
- Use of constraint hierarchies.
- Support for cyclic subsolvers.
- Support for multi-output constraints.

These differences, and their advantages and disadvantages, must be considered when choosing a local propagation algorithm.

One-Way Local Propagation Constraint Solvers

One-way solvers are the simplest type of local propagation solver – they need only determine which constraints should be enforced and in which order. This is because one-way LP solvers can only enforce a particular constraint using one method. For example, consider a constraint such as $x = y + z/2$. This constraint will only ever be enforced by setting x as opposed to setting y or z.

One-way LP solvers represent constraint equations as a directed graph with variables as nodes of the graph and constraints as sets of edges (see Figure 2.1). When the value of a variable is changed, the system can be returned to a state satisfying the constraints by enforcing the constraints in the graph in topological order. Topological sorting of the constraints can be performed using a depth-first search of the constraint graph since the graph is not permitted to contain cycles. The computational complexity of this is $O(V + C)$, where C is the number of constraints in the constraint graph and V is the number of variables.

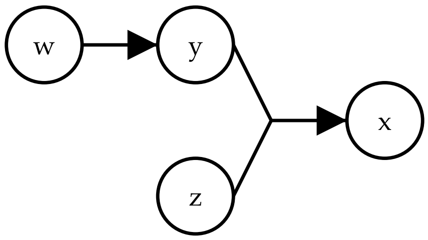


Figure 2.1: Directed constraint graph for constraints $x = y + z/2$, $y = w^2 + 4$ and $z = 5$.

Some one-way local propagation solvers separate constraint solving into two phases – planning (which involves sorting the graph) and execution (which only involves enforcing the constraints). The graph only needs to be sorted when the topology of the constraint graph changes (*i.e.* when constraints are added or removed), whereas enforcing constraints occurs far more often (*i.e.* whenever the user drags a geometric entity). Therefore, if the topologically sorted graph is maintained, enforcing the constraints does not require the graph to be sorted.

Implementation of one-way local propagation solvers is extremely easy, and they can be very efficient. However, they are not very powerful due to the fact that they do not support multi-way constraints, cyclic subsolvers or constraint hierarchies.

Multi-Way Local Propagation Solvers

Multi-way constraints are more general than one-way constraints – they are not limited to invoking one method to enforce a particular constraint. For instance, consider the example used above, $x = y + z/2$. A multi-way constraint solver could satisfy this constraint by either setting x to $y + z/2$, y to $x - z/2$, or alternatively, z to $2 * (x - y)$. This provides the constraint solver (and the application using it) with much greater flexibility. This behaviour is useful in constraint-based CAD since it allows relationships such as *coincident-points* to be specified without having to specify which point should be treated as the input to the constraint and which should be treated as the output.

Like one-way LP solvers, multi-way solvers have to determine the order in which to enforce constraints. However, they also need to determine which method should be used to satisfy each constraint. This is equivalent to transforming the constraint graph from an undirected graph to one that is directed. When directing the graph, the constraint solver must ensure that no variable (*i.e.* node in the graph) has two incoming edges, since this would result in more than one constraint attempting to determine the value of that variable (*i.e.* the system would be over-constrained).

Multi-way solvers have the disadvantage that there may be more than one way to solve any particular set of constraints – the system is under-constrained. The most common way of coping with this problem is by using *constraint hierarchies*. Constraint hierarchies allow constraints to have different levels of importance. Under-constrained systems are handled by over-constraining the system with non-required constraints at decreasing levels of importance. The constraint solver can then satisfy a sufficient number of non-required constraints to ensure that the system is well-constrained.

There are several closely related multi-way local propagation algorithms (see Figure 2.2). These were investigated for purposes of identifying one that would be suitable for use in this project.

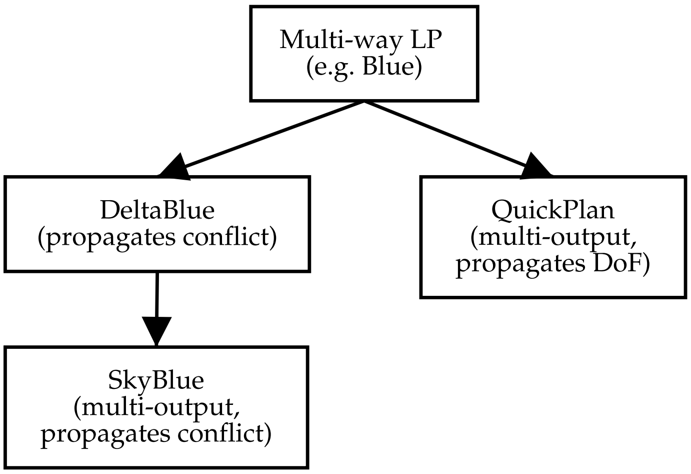


Figure 2.2: Relationship between multi-way LP solvers.

Choice of Algorithm

Solver	Multi-Way	Hierarchies	Cycles	Multi-Output	Complexity
Blue	yes	yes	no	no	$O(V + C)$
DeltaBlue	yes	yes	no	no	$O(V + C)$
SkyBlue	yes	yes	yes	yes	$O(M^C)$
QuickPlan	yes	yes	yes	yes	$O(C^2)$

Table 2.1: Comparison of Multi-Output Local Propagation Constraint Solvers.

From a comparison of Blue, DeltaBlue[28], SkyBlue[27] and QuickPlan[34] (see Table 2.1) it was decided that QuickPlan should be used as the constraint solver for this project.

2.2.4 Visual Language Theory

The creation and manipulation of constraints can be considered to be a form of programming. Much like conventional programming, users of constraint-based CAD packages attempt to achieve an overall goal by instructing the computer to execute sets of operations. In this case each operation is the act of enforcing a particular constraint.

The representation of an individual constraint can be thought of as a *cognitive artifact*: "A cognitive artifact is an artificial device designed to maintain, display or operate upon information in order to serve a representation function."[23] Artifacts should support alteration of the information they represent, and provide the user with feedback on state of the system as a result of the user's actions. An artifact that does not provide a smooth transition between these phases of interaction can disrupt the thought process of the user, causing them to shift their focus of attention from their overall goal to the immediate actions being performed. This disruption can impair performance. One way of attempting to address this issue is to make use of direct manipulation techniques.

Graphical user interfaces, including visual programming environments, often make use of some form of metaphor[4]. The justification behind this is the belief that the use of metaphor enhances usability by assisting the user in the formation of a mental model for use when interacting with the environment in question. However, experimental work[7] has not supported this theory – it is thought that users gain familiarity with an environment through direct manipulation rather than through use of an instructional metaphor.

The crucial attributes of direct manipulation[29] are:

- There should be continuous graphical representation of the objects of interest to the user.
- Manipulation of objects should involve physical actions or labelled button presses.
- Operations performed by the user should be rapid, incremental and reversible.
- The effects of operations on an object of interest should be immediately visible.
- Novices should be able to learn the basic functionality quickly.
- Users should be able to gain access to additional functions as they gain familiarity with the system.

These attributes should be considered when designing the visual representation of constraints.

2.2.5 Data Structure for In-Memory Storage of Geometric Entities

Hierarchical data structures are often used for storing and indexing spatial data. Such data structures are based on the principle of recursive decomposition and make use of spatial indexing, the process of sorting data with respect to the space it occupies.

There are several types of hierarchical data structure that can be used for representing such 2D geometric entities, including plane-sweep methods, point-based methods, line-based methods and area-based methods. However, an investigation of these methods led to the conclusion that for the purpose of this project, none of these data structures would provide a significant performance improvement over an unordered data structure for the following reasons:

- Plane-sweep based methods are most appropriately used if the data is static: insertion of a single item into the data structure can force the re-execution of the algorithm on the entire data set. CAD systems can have highly dynamic data.
- Point-based methods involve representing each geometric object by a point (*e.g.* the centroid). Performing proximity queries can be problematic, unless additional information (such as the entity's bounding box) is stored along with the co-ordinates of the representative point, thus increasing the storage requirements of such a data structure.
- The use of line-based methods, such as the PM quadtree and its variants, has the disadvantage that some operations that can successfully be performed on rectangles cannot be performed on their constituent line segments.
- Area-based methods such as the MX-CIF quadtree, the R-tree and the R^+-tree may reorder the entire data structure upon insertion of an entity. This is too time consuming for use in CAD systems where the location of an entity may change frequently.

It was therefore decided that an unordered data structure, such as `java.util.Vector`, should be used for storage of geometric entities. Whilst this has the disadvantage that search for a particular entity has time complexity $O(N)$, where N is the number of geometric entities in the drawing, insertion and deletion have only $O(1)$ time complexity.

2.2.6 Extensible Markup Language (XML)

XML, a metalanguage for describing constraints on the structure of data, easily satisfies the requirements described in Section 2.1.1. In order to enable correct parsing of data, XML documents must be *well-formed*. For data to be a well-formed XML document, the well-formedness constraints outlined in the XML Specification[14] must be satisfied.

XML documents may also be *valid*. Unlike well-formedness, validity is not required of XML documents. An XML document is valid if it has a document *type definition* (DTD) and the document satisfies the constraints outlined in its DTD. A DTD defines the structure of an XML document by specifying the allowable content of an XML document, such as the structure and nesting of elements, attributes, attribute values and comments. Validating XML can be time consuming, however it was felt that the benefits of validation (*e.g.* self-documentation) outweighed the slight loss of efficiency. An XML document can also be *schema valid* if it is constrained by a *schema* rather than a DTD. However, XML Schema is not part of the XML 1.0 Specification. It was therefore decided that DTDs should be used rather than schemas.

Frameworks for XML Parsing

In choosing a framework for XML parsing, three alternatives were considered:

- SAX, the *Simple API for XML*[19], is an event-based API for parsing of XML data. Applications implement *handlers* to process parsing events. SAX enables very fast, sequential access to XML data.
- DOM, the *Document Object Model*[19] is another XML parsing framework. Unlike SAX, DOM is a standard and covers content models other than XML. DOM is *tree-based*. This means that it builds an in-memory tree representation of the XML document being parsed. Applications can then traverse the tree and modify the data.

- JDOM, the *Java Document Object Model*[21] is a lightweight, Java2-based API for representing a document in Java. It can support any hierarchical data format (but is compliant with current XML standards). JDOM enables reading, writing and manipulation of XML documents using a tree structure. This tree structure can be created using either the SAX or DOM parsing frameworks.

Choice of XML Parsing Framework

Framework	Fast Data Access	Low Mem. Req.	Manipulate Data	Hierarchical Data
SAX	yes	yes	no	no
DOM	no	no	yes	yes
JDOM	yes	yes	yes	yes

Table 2.2: Comparison of XML parsing frameworks.

From a comparison of the advantages and disadvantages of SAX, DOM and JDOM (see Table 2.2), it was decided that use of JDOM would be most appropriate. This is because use of JDOM results in the flexibility provided by DOM, but with the efficiency of using SAX.

XML Parser

SAX, DOM and JDOM are not XML parsers – they are simply frameworks for XML parsing. To be able to parse XML documents, an XML parser is needed. There are several XML parsers that are available in Java, including Apache Software Foundation's Xerces, IBM's XMLJ4 and James Clark's XP. Apache's Xerces was chosen for use in this project.

2.3 Choice of Tools

The project was implemented in Java, using Sun's Java 2 SDK, Standard Edition, version 1.2.2 for Linux. Java was chosen over other object-oriented programming languages because of its support for user interface design (Swing API), and 2D graphics (Java 2D API). Code was written in GNU Emacs (Java mode), compiled using Sun's Java compiler (`javac`) and debugged using the Java debugger (`jdb`). To assist with the compilation process, the GNU `make` utility was used. Prior to writing code for the project, test programs were written to gain familiarity with the Java 2D API and Swing.

I had originally intended to use Thor, the undergraduate teaching system, for development. This proved to be impractical since there was too great a lag in interactive response time. Instead, my personal computer (running Debian GNU/Linux) was used. CVS, a version control system, was set up to store the code, and frequent backups of the entire project were made and copied to Thor and other machines within the university. A shell script (written for the GNU Bourne-Again Shell) was used to automate the backup process.

As described in the previous section, JDOM (the Java Document Object Model) and Apache's Xerces XML parser were used to read, write and manipulate XML data. Having never used JDOM and Xerces before, I wrote test programs to learn more about them prior to commencing the implementation stage of the project.

2.4 Development Process

Two commonly used software development models were considered – *The Waterfall Model* and *Iterative Development* [31, 2].

Although the Waterfall Model is widely considered to be the best approach where applicable due to its structured nature, it was felt that Iterative Development would be more appropriate for this project. The Waterfall Model is rarely used for the development of software with a heavy human–computer interaction emphasis. This is because requirements relating to the human–computer interface cannot necessarily be fully defined prior to development or prototyping. Instead, prototyping is often used to construct a functional user interface for purposes of obtaining user feedback [5]. An iterative approach then can be taken, allowing the prototype to be refined on the basis of user comments until it has achieved full system functionality. Another approach to prototyping advocated by HCI research is that of creating many prototypes, thus enabling the usability of several possible solutions to be evaluated before choosing to develop one particular solution. This was felt to be too time consuming for a project of this scale and so, as a compromise, possible solutions for aspects of the project relating to HCI were discussed with users prior to the implementation of a prototype, thus ensuring that users' opinions were considered.

Once the initial specification of each module was finalised, the development process used consisted of:

- Implementing each module with minimal functionality.
- Testing the module and evaluating those modules relating to human–computer interaction by obtaining user feedback.
- Integrating the module with the rest of the system.
- Testing the system ensuring that the emphasis of the tests used was on the newly integrated module (again obtaining user feedback if the module has a human–computer interaction emphasis).

Additional functionality could be added to the module and the process repeated. A diagram of the process is shown in Figure 2.3.

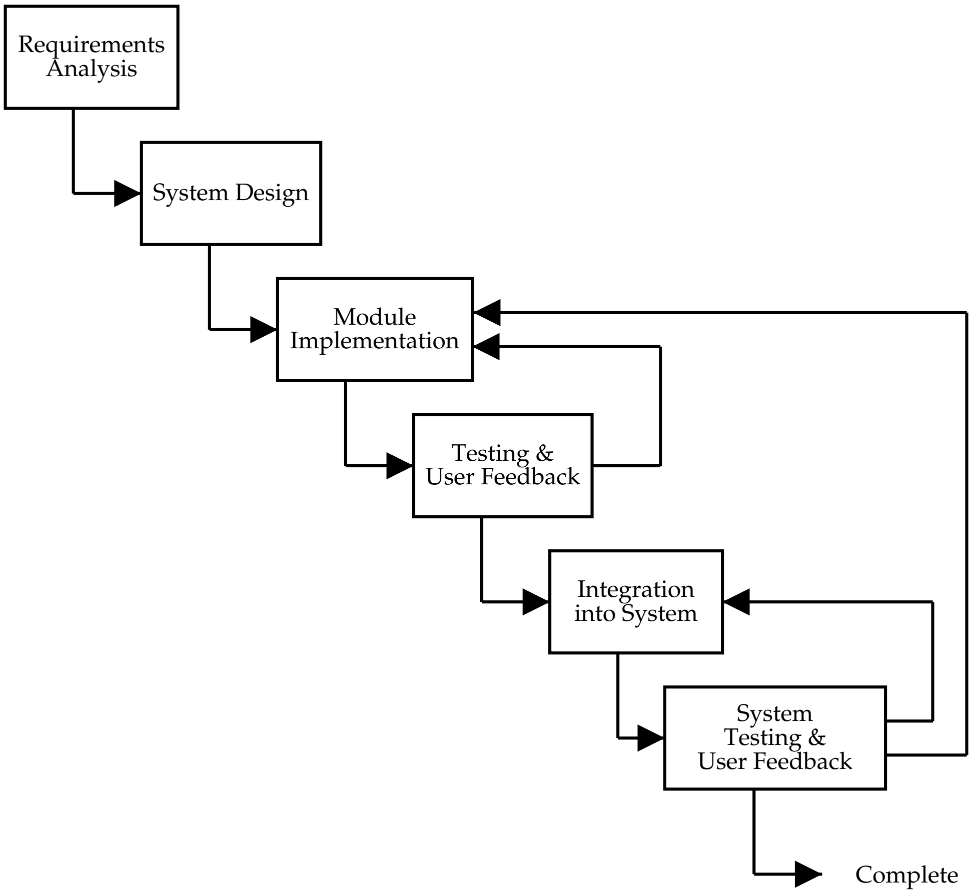


Figure 2.3: Development process.

2.5 Summary

In this chapter, I have described the work undertaken prior to the implementation stage of the project. Requirements were identified, thus refining and clarifying the overall aims of the project. A summary of my research into existing constraint-based CAD systems, constraint solving techniques, visual language theory and XML technologies was given. An introduction to the technical details of local propagation constraint solvers was provided, and specific local propagation algorithms were analysed and compared leading to the choice of a constraint solving algorithm for use in the project. The tools used in the implementation stage of the project were listed, and an explanation of the software development process employed was given.

System architecture and implementation details are described in the next chapter.

3

Implementation

This chapter describes the implementation of a constraint-based drawing system satisfying the requirements outlined in Section 2.1.1. The architecture of the system is described, and implementation detail of each component is given. The components described are (in order of appearance in the chapter) are the constraint solver, the constraints implemented, the drawing package (including integration of the drawing package and constraint solver, visual representation of constraints, the overall user interface, and external data representation.

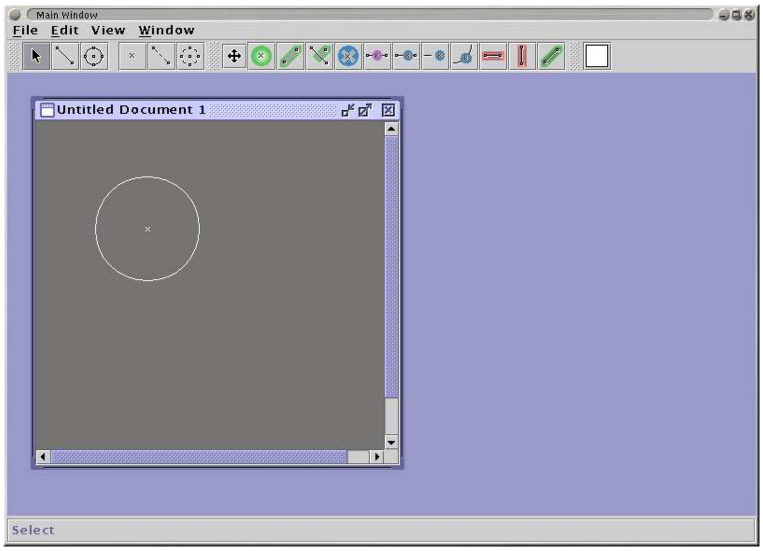


Figure 3.1: User interface of the system.

All components were successfully implemented and integrated to form a fully functional system (the user interface of which is illustrated in Figure 3.1), which was then subjected to usability testing (see Section 4.3).

3.1 System Architecture

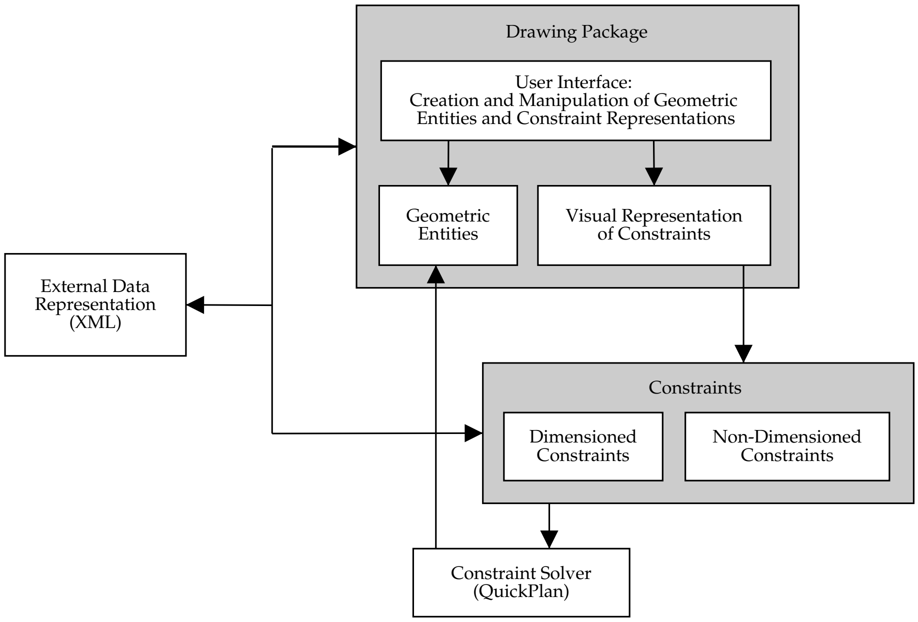


Figure 3.2: System architecture.

From the point of view of the architecture of the system (see Figure 3.2), there are four major components:

- A drawing package.
- A constraint solver.
- Constraints.
- An external data representation.

These components can be further divided according to the view of the system presented to the user – constraints can constrain particular dimensions or enforce a non-dimensioned geometric relationship, and the drawing package can be considered to consist of geometric entities and the constraint representation. These aspects of the drawing package were implemented separately and evaluated using empirical evaluation techniques (see Section 4.2).

3.2 Constraint Solver

As described in the previous chapter, QuickPlan[34] was chosen as the constraint solver to be used in this project. QuickPlan is an incremental algorithm capable of satisfying any hierarchy of multi-way, multi-output constraints that has at least one acyclic solution, in less than $O(C^2)$, where C is the number of constraints.

3.2.1 QuickPlan Algorithm

There are two versions of the QuickPlan algorithm:

- A non-incremental version that examines the entire constraint graph whenever a constraint is added or removed.
- An incremental version that reduces the number of constraints examined (compared with the non-incremental version) whenever a constraint is added or removed, and allows the set of constraints to be edited.

To gain familiarity with the algorithm, both the non-incremental and incremental versions were implemented. Only the incremental algorithm is used in the final code; the operation and implementation of the non-incremental version is not covered here.

The incremental QuickPlan algorithm has two phases:

- A planning phase, based on propagation of degrees of freedom, in which a method is chosen for each constraint.
- An execution phase, in which the methods are executed in topological order.

QuickPlan's Planning Phase

Due the length restrictions on this report, a detailed explanation of the operation of QuickPlan's planning phase is provided in Appendix A.

A `ConstraintGraph` class was implemented to define the methods and data structures required by QuickPlan's planning phase.

QuickPlan's Execution Phase

Before the execution stage of the algorithm, the constraints must be topologically sorted. A topological sort of a directed acyclic graph is an ordering of the vertices of the graph such that if the graph contains an edge from vertex u to vertex v, then u will be before v in the ordering[8].

Methods were written to topologically sort the constraint graph using a depth first search, and return a vector of the topologically sorted constraints (see Appendix F.1). The time complexity of the sort is $O(C + E)$, where C is the number of constraints in the graph, and E is the number of edges.

Adding and Removing Constraints

QuickPlan treats the new constraint to be added as an unenforced constraint. The algorithm for adding constraints (see Appendix B) attempts to satisfy the constraint to be added by identifying whether the constraint has a method that outputs only to the constraint's free variables. If so, this method can be selected to enforce the constraint. If no such method exists, the constraint is added to the queue of unenforced constraints, and the algorithm calls the constraint solver to attempt to enforce the constraint. Note that when attempting to enforce the constraint, only constraints upstream of the constraint to be added will need to be examined (for justification see [34]).

To remove a constraint, QuickPlan removes the constraint from the constraint graph and treats the constraint being removed as a retracted constraint. If the constraint was enforced at the time of removal, the algorithm sets the priority queue of unenforced constraints to contain those constraints that are downstream of the constraint to be removed and of equal or lesser strength. The algorithm then attempts to find a locally-graph-better solution by calling the constraint solver to attempt to enforce these constraints.

Methods implementing the algorithms for adding and removing constraints are defined in the `ConstraintGraph` class.

3.2.2 Data Structures

The incremental version of QuickPlan requires the use of certain data structures to ensure correct operation of the algorithm. Constraints and variables are required to contain certain fields (see Section 3.3), and several global variables and data structures are required, including those in Table 3.1.

Name	Description
cnsToEnforce	constraint that the solver is trying to enforce
retractableCnsQueue	a priority queue of constraints, ordered by increasing strength, that may be retracted in order to enforce another constraint
undoStack	stack for storing information required to undo changes to the constraint graph
unenforcedCnsQueue	priority queue of retracted constraints, ordered by decreasing strength
freeVariableSet	free variable set
potentiallyUndeterminedVars	set of variables that may become undetermined

Table 3.1: Global data structures and variables required by QuickPlan.

The priority queues were implemented using a heap data structure[8]. This involved writing two classes – one implementing a priority queue, `PriorityQueue`, and one implementing a heap, `BinaryHeap`. The constraint with the greatest (or least) strength can be returned in $O(1)$ and removed from the queue in $O(\log N)$ for a N-element heap. Insertion of a new element also takes $O(\log N)$.

Unordered vectors, `java.util.Vector`, were used to implement the sets, and the undo stack was implemented using the `java.util.Stack` class.

3.3 Constraints

This section provides an overview of the constraints implemented. The geometric equations used in implementing the constraints can be found in Appendix C. Each constraint affects one or more `GeometricObjects` (the `GeometricObject` interface is discussed in Section 3.4.1), but all constraints are internally defined in terms of `DoublePoint2Ds`. (This is possible because all other geometric entities are defined by `DoublePoint2Ds`.) In other words, the only geometric entities treated as variables to be explicitly determined by the QuickPlan algorithm are `DoublePoint2Ds`.

QuickPlan requires all variables to be determined by the algorithm to contain certain fields (see Table 3.2). Therefore, the `DoublePoint2D` class (see Section 3.4.1) incorporates these fields, as well as methods for setting and accessing their values.

Name	Description
determinedBy	the constraint that determines the value of this variable
constraints	the set of constraints that make use of this variable
numConstraints	the number of constraints that reference the variable
walkbound	minimum strength upstream constraint that would have to be retracted to allow the variable to be determined by a different constraint
mark	used to mark the variable as visited

Table 3.2: Fields that the QuickPlan algorithm requires each variable to contain.

All the constraints implemented are subclasses of `Constraint`, an abstract class containing the fields required by the QuickPlan algorithm (see Table 3.3), as well as methods for setting and accessing each of the fields.

Name	Description
variables	the set of variables referenced by this constraint
methods	the set of methods used to enforce this constraint
selectedMethod	the method that currently currently satisfies the constraint
strength	the constraint's strength in the constraint hierarchy
mark	used to mark a constraint as visited

Table 3.3: Fields that the QuickPlan algorithm requires to be associated with each constraint.

Each `Constraint` contains one or more member classes that extend an abstract `ConstraintMethod` class. These member classes represent the methods that can be used to enforce that `Constraint`. A constraint with multiple `ConstraintMethods` is a multi-way constraint. Where possible, `Constraints` are multi-way constraints to increase the number of possible solution graphs that can be constructed, thus increasing flexibility.

Eleven types of constraint were implemented (see Table 3.4), some of which constrain particular dimensions, and some of which enforce non-dimensioned geometric relationships.

The user can create additional constraints by combining constraints. For instance, the radius of a `DoubleCircle2D` can be constrained by using a `PointOnCircleConstraint` to constrain a `DoublePoint2D` to lie on the circumference of the `DoubleCircle2D`, and then constraining the distance between the `DoublePoint2D` and the centre of the `DoubleCircle2D`.

Class Name	**Description of Constraint**
`PositionConstraint`	Constrains a `DoublePoint2D` to be at a given location.
`DistanceConstraint`	Constrains the distance between two `DoublePoint2D`s.
`PointLineDistanceConstraint`	Constrains the distance between a `DoublePoint2D` and a `DoubleLine2D`.
`LineAngleConstraint`	Constrains the angle of inclination of a `DoubleLine2D`.
`CoincidentPointsConstraint`	Constrains two `DoublePoint2D`s to be coincident.
`MidPointLineConstraint`	Constrains a `DoublePoint2D` to lie at the mid-point of a `DoubleLine2D`.
`PointOnLineConstraint`	Constrains a `DoublePoint2D` to lie on a `DoubleLine2D`.
`PointOnLineExtendedConstraint`	Constrains a `DoublePoint2D` to lie on the the extension of a `DoubleLine2D`.
`PointOnCircleConstraint`	Constrains a `DoublePoint2D` to lie on the circumference of a `DoubleCircle2D`.
`HorizontalLineConstraint`	Constrains a `DoubleLine2D` to be horizontal.
`VerticalLineConstraint`	Constrains a `DoubleLine2D` to be vertical.

Table 3.4: The constraints implemented.

3.4 Drawing Package

The drawing package is a subsidiary part of the project, however, it was necessary to provide sufficient complexity to demonstrate the power of the QuickPlan algorithm and the operation of my constraint representation. This section describes the geometric primitives supported by the drawing package, as well as the functionality provided for creating, selecting and manipulating geometric entities. Integration of the drawing package and constraint solver is also described.

The majority of the functionality of the drawing package is contained within the `Drawing` class which is an extension of `javax.swing.JPanel`, a general purpose lightweight component provided by Java's Swing as a replacement for the Abstract Windowing Toolkit's `java.awt.Panel` and `java.awt.Canvas`. The `Drawing` class defines over fifty methods that provide functionality for creation, selection and manipulation of geometric entities and constraints, as well as methods for displaying geometry and visual representations of constraints.

Methods inherited from the `javax.swing.MouseInputListener` interface are used to detect mouse events. These methods are used to invoke other methods to perform tasks such as setting the tool currently being used, and selection, creation and manipulation of geometric entities and constraints.

3.4.1 Geometric Entities

Geometric entities can have one of two functions:

- *Primary entities* are components of the final drawing.
- *Construction entities* are not part of the final drawing, but are used during the process of constraining the primary entities.

The geometric entities provided are points, line segments and circles. Line segments and circles can be either primary or construction entities. Construction line segments and circles are visually distinguished from their primary counterparts by being displayed using dashed lines (see Figure 3.3). Points are only meaningful as part of a drawing when considered in relation to another object, and are therefore always treated as construction entities. The user can toggle visibility of construction entities.

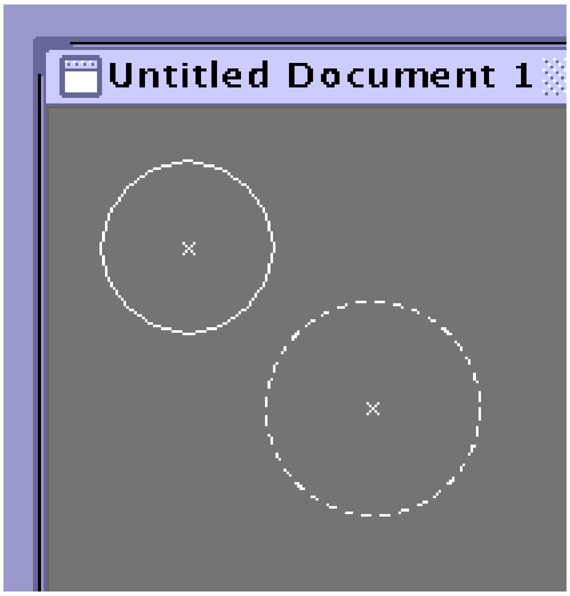


Figure 3.3: Primary and construction circles.

To express the distinction between general methods of any geometric entity and methods specific to a particular type of geometric entity, as advocated by Stroustrup[32], a `GeometricObject` interface was written:

```
public interface GeometricObject {
    public void setColor(Color c);
    public Color getColor();
    public DoublePoint2D[] getPoints();
```

```
    public boolean isConstructionObject();
    public Rectangle2D.Double[] getSelectionHandles();
    public boolean isMouseClickedOn(Point2D clickPt);
    public void moveDefiningPt(DoublePoint2D pt, double x, double y);
    public void translate(double x, double y);
    ...
}
```

This interface is implemented by the classes defining the three types of geometric objects provided – `DoublePoint2D`, `DoubleLine2D`, `DoubleCircle2D`.

Points

In addition to implementing the `GeometricObject` interface, points (`DoublePoint2D`) are a subclass of `java.awt.geom.Point2D.Double`. The coordinates of the point are stored using `double` fields (IEEE 754 64-bit floating-point): CAD drawings often require double precision to be used to represent numbers to sufficient accuracy.

Line Segments

Line segments (`DoubleLine2D`) implement the `GeometricObject` and `java.awt.Shape` interfaces. Each line segment is defined by two `DoublePoint2D`s. Since constraints operate upon `DoublePoint2D`s (see Section 3.3), the location and orientation of each line segment is defined by any constraints currently determining the positions of its end-points. Each `DoubleLine2D` is either a primary or construction line segment.

Circles

As well as implementing the `GeometricObject` interface, circles (`DoubleCircle2D`) extend `java.awt.geom.RectangularShape`, an abstract class used as a base class for `java.awt.Shape` implementations that have a rectangular bounding box. Circles are defined by two `DoublePoint2D`s – the centre of the circle, and a point on the circumference of the circle that defines the radius of the circle.

The existence of the radius point is not revealed to the user, since its sole purpose is in enabling a larger class of constraints to be applied to circles than would otherwise be possible. Like line segments, each `DoubleCircle2D` can be a primary or construction circle.

3.4.2 Line Colour

A tool is provided for changing the line colour for new geometric entities (see Figure 3.15. When the user selects this tool, a colour chooser is displayed (see Figure 3.4), from which a new colour may be chosen.

3.4.3 Input Model

Input options are indicated to the user by two toolbars (see Figure 3.5) – one for geometric entities, and one for constraints. Tools are provided for the creation and manipulation of primary entities (line segments and circles), construction entities (points, line

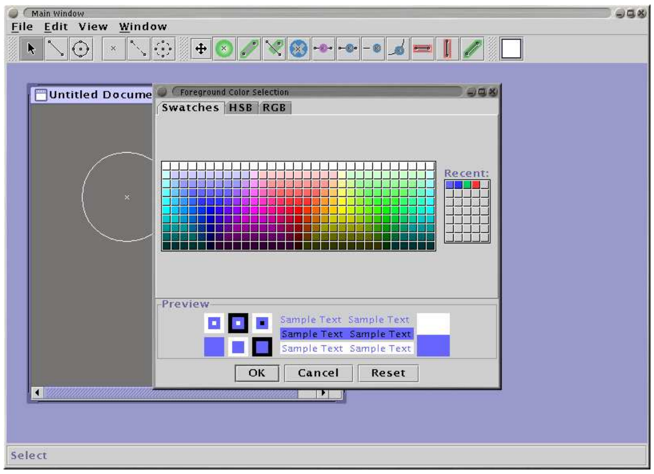


Figure 3.4: Colour chooser.

segments and circles) and constraints (for discussion of the creation and manipulation of constraints, see Sections 3.5.7 and 3.5.8). The `javax.swing.ButtonGroup` class is used to ensure that selection of these tools is mutually exclusive.

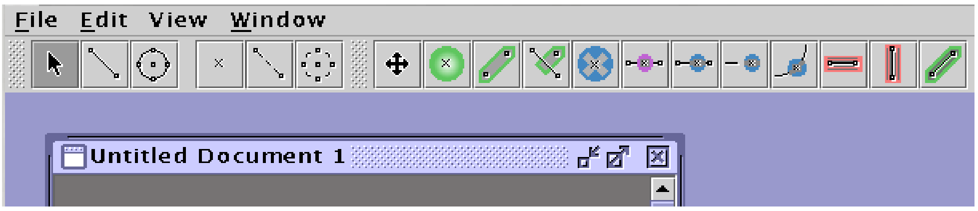


Figure 3.5: Geometric entities toolbar (left) and constraint toolbar (right).

Input of a particular geometric entity is carried out by first selecting the appropriate tool from the geometric entities toolbar. When the user starts creating an entity, the current selection is set to the entity being created. Input is performed using either a *click* mechanism for points, and a *press-drag-release* mechanism for line segments (see Figure 3.6) and circles (see Figure 3.7).

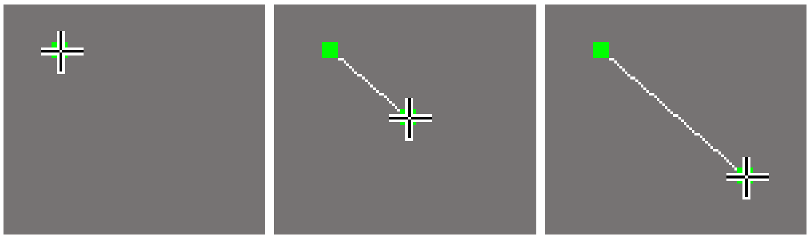


Figure 3.6: Creation of a line segment: Press, drag, release.

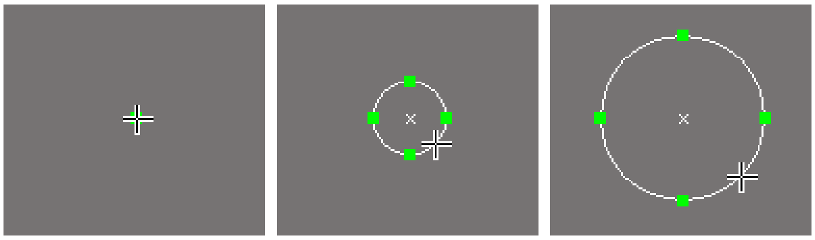


Figure 3.7: Creation of a circle: Press, drag, release.

The use of a press-drag-release mechanism provides consistency between the creation of a line segment or circle and subsequent translation or resizing of the segment or circle, which is also performed using a press-drag-release mechanism. Construction line segments and circles are input in exactly the same manner as their primary counterparts.

If the user creates a line segment of zero length, or a circle of zero radius, the line segment or circle is not added to the drawing. This prevents the user from accidentally creating lines or circles which are too small to be manipulated.

3.4.4 Selection Model

The entities that can be selected are points, line segments, end-points of line segments, and circles and their centre points. Selected entities are visually distinguished from entities not part of the current selection using *selection handles* (shown in green in Figures 3.6 and 3.7).

Standard selection behaviour is implemented. Clicking the mouse button while the cursor is near to an entity sets the current selection to that entity. Selection of multiple entities can be performed by creating a new selection using a *selection rectangle*. To add or remove an entity from the current selection, the shift button must be held down and the mouse button clicked while the cursor is near to the entity. The current selection is cleared if the primary mouse button is clicked while the mouse cursor is *not* on, or near, any entity.

The selection process is initiated by the `mousePressed` method whenever the user clicks the primary mouse button or starts dragging the mouse whilst using the selection tool.

A `GeometricObject` variable is used to store a reference to the entity most recently clicked on. An unsorted vector of references to currently selected entities is also maintained. Each reference either points to an entity in the geometric entities vector, or to a defining point of one of these entities. This vector can be accessed by other methods within the drawing canvas.

Points are considered to be "above" other geometric entities, because the area occupied by the on-screen representation of a point is less than the area occupied by other geometric entities.

Threaded Architecture for Selection of Multiple Geometric Entities

A novel approach was used to improve responsiveness to selection of multiple geometric entities. Once the user releases the primary mouse button, indicating that they have finished positioning the selection rectangle, an instance of a subclass of `java.lang.Thread` is created for each geometric entity in the drawing to determine whether the entity falls within the selection rectangle. If the entity is contained within the selection rectangle, a reference to the entity is added to the vector recording the current selection. Synchronisation problems do not occur, because `java.util.Vector`'s methods for adding elements to the vector are marked with the `synchronized` keyword, so that they can only be invoked by one thread at a time. This use of threads resulted in a noticeable performance improvement over a non-threaded method of determining which entities lie within the selection rectangle.

3.4.5 Manipulation of Geometric Entities

Manipulation of geometric entities is performed using the select tool. The selection process is invoked prior to manipulation of entities. This has the effect of ensuring that the entity to be manipulated is part of the current selection.

A point or line segment is translated by dragging it to the desired location. Circles are translated by dragging their centre point to the desired location. If several entities are already selected, and the user starts dragging the mouse while the cursor is over an entity that is part of the current selection, the entire selection will be translated.

Resizing of a line segment is performed by dragging one of its end points to a new location. A circle can be resized by dragging its circumference.

3.4.6 Integration of Constraint Solver and Drawing Package

Prior to integrating the constraint solver with the drawing package, two constraints were created – a position constraint and a distance constraint (see Section 3.3). These constraints were used for testing during the integration process.

Stay Constraints

When the constraint solver and drawing package were integrated, it became apparent that the system was not conforming to the *principle of least astonishment*[3]. In other words, from the point of view of the user, the system was not behaving in a predictable fashion.

For example, consider a multi-way constraint for constraining the distance between two points, $P1$ and $P2$. Such a constraint will have two satisfaction methods, enabling the location of point $P1$ to be based on the location of $P2$ or vice versa. When such a constraint is added to the system, the solver will direct the constraint graph so that one of these methods is being used to enforce the constraint. Since the constraint graph is redirected only when constraints are added or removed from the system, provided no other constraints are determining the location of either $P1$ or $P2$, when one of these points is dragged the user will not know in advance whether the point being dragged is being constrained to lie within a certain distance of the other point, or vice versa. The effect of this is that sometimes the point being dragged will only be allowed to move in a circle with radius equal to the distance that the constraint specifies, and other times, the point will be allowed to be dragged to wherever the user desires, and the other point will be constrained to follow it.

To eliminate this ambiguity, an explicit stay constraint of a weak strength is added to the defining points of any geometric entity that is being manipulated at the start of manipulation, and removed when manipulation finishes. It redirects the constraint graph so that any constraints that attached to points defining the entity being manipulated will (if possible) treat the points as the input to the constraint. This ensures that behaviour of constraint entities during manipulation is consistent.

3.5 Visual Constraint Representation

This section describes the implementation of visual representations of each of the constraints in Section 3.3, as well as the functionality provided for constraint creation and manipulation.

A number of possible design solutions were initially explored. The solution felt to be the clearest was inspired by research on the use of colour for display of layered information in air traffic control (ATC) displays[26]. The underlying concept is to represent constraints as coloured infilled areas bounding the geometry they constrain, with different fill patterns to distinguish between different constraint types.

A primary concern in designing a visual constraint representation is visual emphasis – geometric entities should have a greater visual emphasis than the constraint representation. One way of thinking about this is to consider geometric entities to be in the "foreground" and constraints in the "background". This distinction is enforced by the choice of colours, as well as techniques such as blurring. To ensure that no constraint is obscured by another constraint, constraint representations are translucent.

Each subclass of `Constraint` contains a method (`drawConstraint`) for drawing its visual representation. The `java.awt.Graphics2D` context of a `Drawing` can be passed to this method, and the visual representation will be drawn on it.

3.5.1 Layering and Transparency

The appearance of transparency is achieved by blending "transparent" pixels with pixels that the transparent pixels are to be painted over. Blending is performed by using an equation on the red, green and blue components of the colours separately. The exact form of the equation depends on the *compositioning rule* used.

Java supports a subset of the Porter Duff compositioning rules. These rules describe different ways of blending a source colour, CS, with a destination colour, CD, to produce a new destination colour, CD'. Each rule is of the form:

$$CD' = CS \times FS + CD \times FD \tag{3.1}$$

where FD and FS are the fractions of CS and CD to be used in the blending operation respectively. Each Porter Duff rule uses a different pair of values for FS and FD.

To achieve a transparent layered effect when displaying constraint representations, the *Source Over* rule is used in rendering the each constraint representation onto a `Drawing`.

$$CD' = CS \times \alpha + CD \times (1 - \alpha) \tag{3.2}$$

where α is the alpha value (between 0 and 1) of the source colour, CS. This rule draws the source colour over the destination colour by combining the source and destination colours based on the transparency of the source colour.

When drawing constraint representations, an alpha value of 0.45 is used. The alpha value is reset to 1.0 prior to drawing geometric entities.

3.5.2 Blurring

To further distinguish between geometry and constraints, each constraint representation is blurred prior to displaying it on a `Drawing` (see Figure 3.8). This reinforces the notion of constraints as part of the background, and geometry as the main focus of attention.

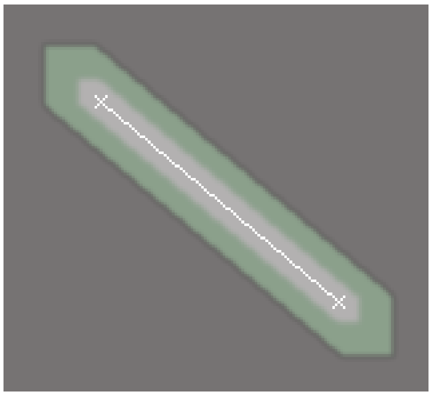


Figure 3.8: Blurred constraint representation.

Blurring is performed using a *convolution filter*. Filtering is a linear operation, implemented by convolving an image with one or more *filter kernels* to produce a new image. In 2D, the discrete convolution of an image array, $f(x, y)$, with a 2D filter kernel, $g(x, y)$, can be represented by the following equation:

$$f'(x,y) \; = \; \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} g(i,j) \times f(x-i, y-j) \tag{3.3}$$

A 3 × 3 Gaussian blurring filter (see Table 3.5) was used to blur the visual representation of each constraint. Gaussian blurring is generally smoother than basic blurring, in which the values in the filter kernel are identical. To ensure that the brightness of the constraint representation is unaltered by the blurring process, the values in the filter kernel sum to 1.

$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$
$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{8}$
$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$

Table 3.5: 3 × 3 Gaussian blurring filter.

Java's `java.awt.image.ConvolveOp` and `java.awt.image.Kernel` classes provide support for convolving `java.awt.image.BufferedImages` with filter kernels.

3.5.3 Colours

The colours used for the constraint representations were chosen to have a medium luminance and low saturation so that their visual emphasis is not significantly different from that of the main background colour (see Section 3.16). This use of colour also ensures that the constraint representations do not visually compete with the display of geometric entities. Four colours were chosen (see Figure 3.9).

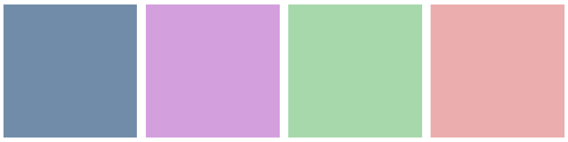


Figure 3.9: Colours used for representing constraints.

The colour used for each constraint type depends on the nature of that constraint type. For instance, all constraints representing a "point-on" relationship are blue.

3.5.4 Shapes

The concept underlying the shapes used is that of *spatial boundedness* – the geometry being constrained is contained within the area of its constraint representation.

Four types of shape are used – circles (see Figures 3.10, 3.12 and 3.14), rectangles (see Figure 3.13), and two types of polygon (see Figure 3.11). The `java.awt.geom.Ellipse2D.Double` and `java.awt.geom.Rectangle2D.Double` classes were used to draw circles and rectangles respectively. The polygons were created using the `java.awt.geom.Area` and `java.awt.geom.GeneralPath` classes.

3.5.5 Fill Patterns

Representations of the different types of constraint (see Section 3.3) are visually distinguished from one another by the use of fill patterns. For dimensioned constraints, the fill patterns also indicate the current value of the dimension being constrained.

Location of a Point

The visual representation of a `PositionConstraint` displays a pale grey circle surrounding the location of the `DoublePoint2D` being constrained (see Figure 3.10).

Distance Between Two Points or a Point and a Line Segment

The visual representation of `DistanceConstraints` and `PointLineDistanceConstraints` has a fill of the same shape as the constraint representation itself, that indicates the length of the distance being constrained (see Figure 3.11).

Figure 3.10: `PositionConstraint`.

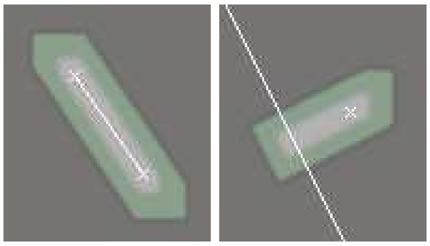


Figure 3.11: From left to right: `DistanceConstraint` *and* `PointLineDistanceConstraint`.

Coincident Points

The representation of each `CoincidentPointsConstraint` constraint contains a white cross (see Figure 3.12), distinguishing it from constraints that constrain a `DoublePoint2D` to lie on a `DoubleLine2D` or `DoubleCircle2D`. The cross shape is the union of two rectangular `java.awt.geom.Areas`.

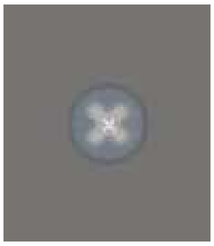


Figure 3.12: `CoincidentPointsConstraint`.

Angle of Inclination of a Line Segment

`HorizontalLineConstraint`, `VerticalLineConstraint` and `LineAngleConstraint` all make use of the `java.awt.TexturePaint` class to create striped fills inclined at the same angle as the line segment being constrained (see Figure 3.13).

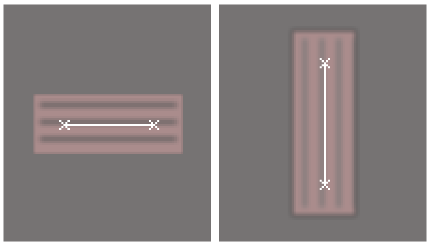


Figure 3.13: From left to right: `HorizontalLineConstraint` *and* `VerticalLineConstraint`*.*

Point on Line Segment or Circle

The `PointOnLineConstraint`, `MidPointLineConstraint`, `PointOnLineExtendedConstraint` and `PointOnCircleConstraint` are all represented by a small pale grey square at the location of the `DoublePoint2D` being constrained, and a white line or arc indicating the direction(s) in which the the point is able to move (see Figure 3.14).

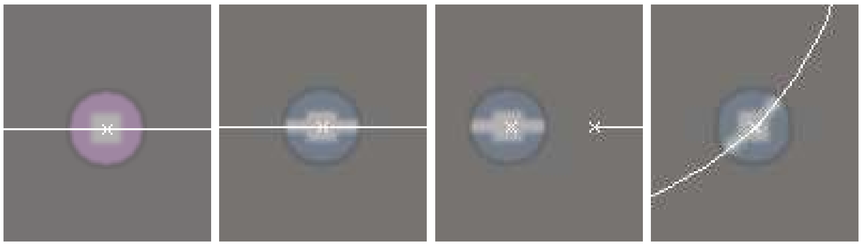


Figure 3.14: From left to right: `MidPointLineConstraint`, `PointOnLineConstraint`, `PointOnLineExtendedConstraint` *and* `PointOnCircleConstraint`*.*

3.5.6 Creation and Manipulation of Constraints

Creation and manipulation of constraints are performed using direct manipulation. This enables similar gestures to be used for creation and manipulation of constraints and geometric entities, ensuring consistency. In addition, the use of direct manipulation of constraint representations provides users with continuous feedback on their actions, resulting in a smooth transition between the alteration and evaluation phases of interaction, as advocated by Norman[23].

3.5.7 Constraint Creation

Constraint creation is carried out by first selecting the appropriate tool from the constraint toolbar (see Figure 3.5), and then selecting the entities to be constrained by clicking the mouse near to the each of these entities. The constraint solver then attempts to add the desired constraint. If this is not possible, a message dialog is displayed indicating this.

To adhere to the principle of least astonishment, geometric entities are moved as little as possible when the constraint is added – for instance, if a `DistanceConstraint` is added to constrain the distance between two `DoublePoint2Ds`, the constraint added will constrain the distance to be the current distance between them. The user can modify this distance by directly manipulating the constraint representation at a later stage.

3.5.8 Constraint Manipulation

The visual representation of constraints that have a particular value associated with them (`PositionConstraint`, `DistanceConstraint` and `LineAngleConstraint`) can be directly manipulated to change this value. Manipulation is performed with the constraint selection tool using a press-drag-release mechanism.

If the cursor is moved over the representation of a constraint that can be manipulated, the cursor changes the default cursor to the move cursor. A subclass of `java.lang.Thread` is used to determining whether the cursor is over the representation of a constraint that can be manipulated.

3.6 User Interface

This section describes the overall user interface of the program. Decisions regarding the design of the user interface were justified using empirical evaluation techniques (see Section 4.2). Figure 3.15 is a screenshot of the user interface. Important features of the user interface are annotated.

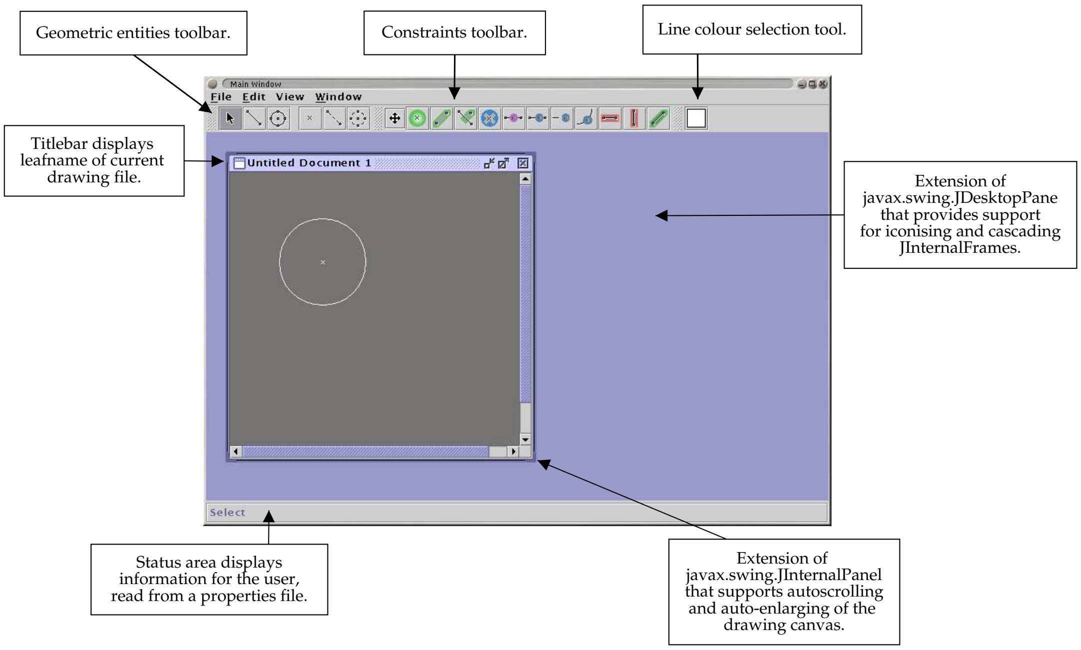


Figure 3.15: Screenshot of the user interface.

3.6.1 Undo/Redo

Each drawing makes use of Swing's `UndoManager` and `UndoableEditSupport` classes to provide support for up to a hundred undoable and redoable operations (such as creation of geometric entities and constraints).

3.6.2 View Modes

Traditionally, display of CAD drawings involves coloured lines on a black background or black lines on a white background. This use of colour clearly highlights the geometry displayed, but can be visually tiring to work with. It was therefore decided that the user should be provided with three viewing modes – black-on-white mode, colour-on-black mode and constraint mode (see Figure 3.16). Constraints are not displayed in black-on-white and colour-on-black modes. If the user selects a constraint creation tool while in black-on-white or colour-on-black mode, the view mode is changed to constraint mode.

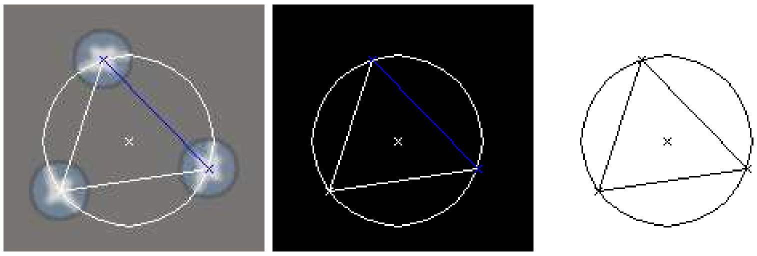


Figure 3.16: From left to right: constraint mode, colour-on-black mode, black-on-white mode.

The background colour for the constraint mode was chosen to be a medium luminance, low saturation grey[26]. This colour is visually restful yet enables geometric entities to be clearly discerned.

Figure 3.17: Background colour.

3.6.3 Visibility of Construction Entities

Visibility of construction entities can be toggled using a check box in the view menu. When visibility of construction entities is disabled, constraints that affect only construction entities are not displayed either. If the user selects a construction entity creation tool whilst visibility of construction entities is disabled, visibility is automatically re-enabled.

3.6.4 Visibility of Constraints

When the drawing is in the constraint view mode, visibility of particular constraint types can be disabled by ticking the appropriate check box in the view menu. If the user selects a particular constraint creation tool whilst visibility of that constraint type is disabled, visibility of that constraint type is automatically re-enabled. When the drawing is in either of the black-on-white and colour-on-black view modes, the constraint visibility check boxes are disabled, since no constraints are visible in these modes.

3.7 External Data Representation

In order to externally represent a drawing in such a way that it can be successfully recreated, the following information is required:

- The width and height of the drawing.
- A description of each `GeometricObject` in the drawing.
- A description of each `Constraint` in the drawing.
- Knowledge of which `GeometricObject` are constrained by which `Constraints`.

3.7.1 Document Type Definition

Document Type Definitions define the structure of an XML document by specifying allowable content, such as the structure and nesting of elements, attributes, attribute values and comments. A DTD was written to define the structure of the XML data to be used by the program. The elements defined include:

```
<!ELEMENT Drawing ( GeometricObjects, Constraints )>

<!ELEMENT GeometricObjects ( DoublePoint2D |
                             DoubleLine2D | DoubleCircle2D )*>

<!ELEMENT DoublePoint2D EMPTY>
<!ELEMENT DoubleLine2D (DoublePoint2D, DoublePoint2D)>
<!ELEMENT DoubleCircle2D (DoublePoint2D, DoublePoint2D)>

<!ELEMENT Constraints ( PositionConstraint | DistanceConstraint |
                        CoincidentPointsConstraint |
                        HorizontalLineConstraint |
                        VerticalLineConstraint | LineAngleConstraint |
                        PointOnLineConstraint |
                        PointOnLineExtendedConstraint |
                        PointOnCircleConstraint |
                        MidPointLineConstraint |
                        PointLineDistanceConstraint |
                        PerpendicularLinesConstraint )*>
```

```
<!ELEMENT PositionConstraint EMPTY>
<!ELEMENT DistanceConstraint EMPTY>
<!ELEMENT CoincidentPointsConstraint EMPTY>
...
```

(Attribute definitions have been omitted for clarity.)

The element definitions form a hierarchy. For example, the element representing the set of geometric entities must contain zero or more elements representing `DoublePoint2Ds`, `DoubleLine2Ds` or `DoubleCircle2Ds`. The order of elements in the DTD is significant. For example, each document must start with a Drawing element – a document that starts with a Constraints element is invalid.

The DTD also defines attributes for certain elements. Attribute definitions are of the form:

```
<!ATTLIST [Enclosing Element]
          [Attribute Name] [type] [Modifier]
          ...
>
```

Each attribute has a type and a modifier associated with it. Attribute modifiers describe the behaviour of the attribute. For example, the attribute definitions for DoublePoint2D elements and PositionConstraint elements are:

```
<!ATTLIST DoublePoint2D
          x CDATA #REQUIRED
          y CDATA #REQUIRED
          color CDATA #REQUIRED
          id ID #REQUIRED>
```

and

```
<!ATTLIST PositionConstraint
          x CDATA #REQUIRED
          y CDATA #REQUIRED
          strength ( 3 | 2 | 1 | 0 ) #REQUIRED
          point IDREF #REQUIRED>
```

To represent which geometric entities are constrained by which constraints, each `DoublePoint2D`, `DoubleLine2D` and `DoubleCircle2D` element is given a unique identifier. This identifier can then be used as a label for that element within the document. Each element representing a constraint has one or more attributes representing the `GeometricObjects` constrained. These attributes are of type `IDREF` – they refer to the `IDs` of the elements representing the constrained `GeometricObject`. This use of unique identifiers and identifier references enables the structure of the `ConstraintGraph` associated with the `Drawing` represented by this document to be recreated.

3.7.2 JDOM

JDOM operates by turning an XML document into a JDOM `org.jdom.Document` object. JDOM `org.jdom.Document` objects are created using a *build tool.* Build tools use an XML parser and either the SAX or DOM API to create a JDOM document. The build tools provided are `org.jdom.SAXBuilder` and `org.jdom.DOMBuilder`. Once a `org.jdom.Document` object has been created, it is not tied to the build tool and parser that built it. In general, `org.jdom.SAXBuilder` is used to create `org.jdom.Document` objects. This is because current DOM parser implementation use SAX to create the DOM tree. The use of `org.jdom.DOMBuilder` to create a `org.jdom.Document` object will always be slower than using `org.jdom.SAXBuilder`. In addition, due to the requirement that the DOM tree must remain in memory for the entire duration of its use (in this case, the conversion process to a JDOM `org.jdom.Document` object, using `org.jdom.DOMBuilder` will also consume more memory. It was therefore decided that `org.jdom.SAXBuilder` would be used as the build tool, and `org.jdom.XMLOutputter` as the output tool (see Figure 3.18).

Figure 3.18: Flow of XML data.

3.7.3 Handling XML Documents

A `XMLHandler` class was written to define methods for converting a `Drawing` to an XML document and vice versa.

Reading XML Documents

To convert an XML document to a `Drawing`, the document is parsed and built into an `org.jdom.Document`. The root element of this `org.jdom.Document` is then passed to a method (`processFile`) which traverses the `org.jdom.Document`. If an element representing a `DoublePoint2D`, `DoubleLine2D` or `DoubleCircle2D` is encountered, the attribute values associated with that element are used to recreate the `DoublePoint2D`, `DoubleLine2D` or `DoubleCircle2D`. So that constraints can be correctly recreated, each `GeometricObject` is stored in a `java.util.Hashtable`, with its unique identifier attribute value as a key. If an element representing a `Constraint` is encountered, the value of its identifier reference attribute values can be used to retrieve the `GeometricObjects` constrained by this constraint from the `java.util.Hashtable`. The `Constraint` can then be recreated correctly.

Writing XML Documents

Converting a `Drawing` to an XML document involves building a `org.jdom.Document` and then adding elements to it corresponding to the `GeometricObjects` and `Constraints` in the `Drawing`. When an element corresponding to each `GeometricObject` is added to the `org.jdom.Document`, the unique identifier for that element is recorded in the `saveID` field of the `GeometricObject`. Since elements representing `Constraints` are added to the `org.jdom.Document` after all `GeometricObjects`, the identifier reference attributes of each element representing a `Constraint` can be set to the unique identifiers of the elements representing the `GeometricObjects` constrained by that `Constraint` by simply retrieving the values stored in the `saveID` fields of the `GeometricObjects`. Once elements corresponding to each of the `GeometricObjects` and `Constraints` in the `Drawing` have been added to the `org.jdom.Drawing`, an instance of the `org.jdom.XMLOutputter` class is used to output the `org.jdom.Drawing` as an XML file.

Saving Files

When the user selects either the "Open" or "Save As" menu item from from the File menu, a file chooser with a default file filter for XML files is displayed (see Figure 3.19).

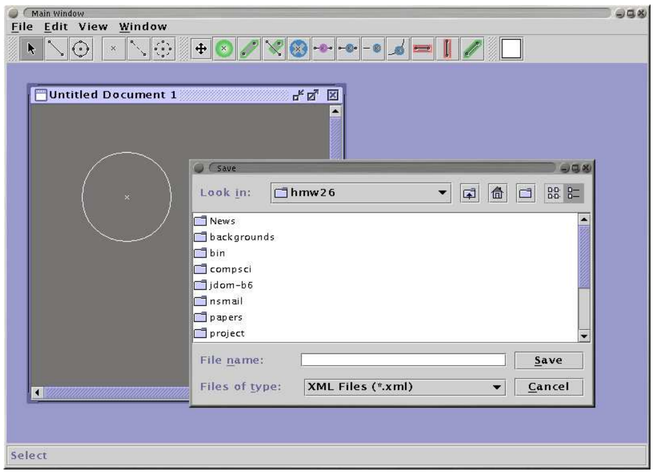


Figure 3.19: File chooser.

If the user tries to close an unsaved drawing, or quit the program while unsaved drawings are open, a dialog box is displayed to confirm the operation so that data is not accidentally lost.

3.8 Implementation Achieved

All the components (see Section 3.2) were successfully implemented and integrated with each other. The system is fully functional as a simple constraint-based drawing package, and was therefore subjected to controlled usability testing (see Section 4.3).

3.9 Summary

In this chapter, I have described the work undertaken in implementing a constraint-based drawing package satisfying the requirements of Section 2.1.1, as well as the requirements outlined in my project proposal. The system architecture was outlined, and each component of the system (constraint solver, constraints, drawing package, visual representation of constraints, user interface, and external data representation) was described.

Testing and evaluation of the system are described in the next chapter.

4

Evaluation

This chapter provides an overview of the types of tasks the system is capable of performing, as well the techniques used to test the system. Throughout the implementation stage of the project, usability of system components with a human–computer interaction emphasis was tested using empirical evaluation. An overview of the evaluation techniques employed is given in this chapter. In addition to this, a description is provided of the controlled experiment designed to evaluate the usability of the completed system.

4.1 Testing

Throughout the implementation stage, testing was performed as often as possible. Each component of the system was tested both individually, and when the component was integrated with the rest of the system.

4.1.1 Drawing Package and Visual Representation of Constraints

The drawing package and visual representation of constraints were developed by iteratively refining a prototype to enable user feedback to inform the implementation process. User feedback was obtained throughout the development process using empirical evaluation (see Section 4.2).

4.1.2 Constraint Solver and Constraints

To facilitate testing my implementation of the QuickPlan algorithm included the ability to generate output corresponding to each stage of the algorithm. This allowed me to test that the behaviour exhibited by my implementation was consistent with the description of the QuickPlan algorithm in[34].

The constraint solver has no problem handling well-constrained and under-constrained systems of constraints. Examples of the type of geometric figures that can be created are shown in Figures 4.1, 4.2 and 4.3. These examples all involve several constraints of various types – Figure 4.1 involves nine constraints, Figure 4.2 involves twenty-seven, and Figure 4.3 involves fifteen.

Constraints were tested in conjunction with the constraint solver. A test harness was written to enable constraints to be added and removed from geometric entities prior to the

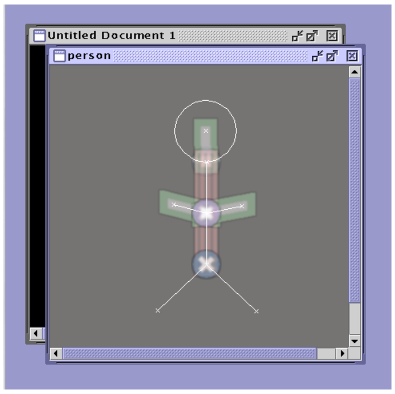


Figure 4.1: The body is constrained to be vertical, and the arms are of fixed length and can be moved as if they were waving.

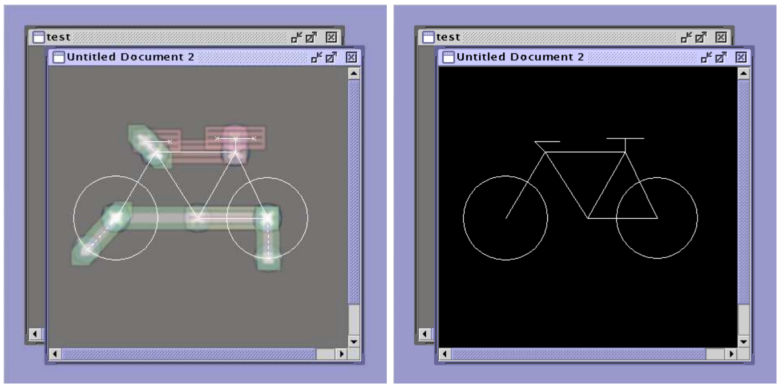


Figure 4.2: Bicycle, viewed in both constraint mode (showing representations of many constraints) and colour-on-black mode (with visibility construction entities turned off).

implementation of the visual representation of constraints. This enabled the interaction between constraints and geometry to be observed. In addition to testing the behaviour of each constraint in isolation, each constraint was tested in combination with other constraints, to determine whether the interaction between constraints was as expected.

4.1.3 External Data Representation

To test the ability of the system to save and load drawings, drawings were created, saved, opened, and saved again. As well as checking that the appearance of the drawing was the same when reloaded, the two saved files were compared to ensure that identical output had been produced, and thus that both loading and saving of files were functioning consistently. All geometric entities and constraint types were included in the drawings used in this test.

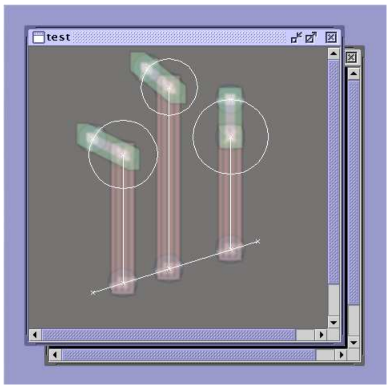


Figure 4.3: The trees remain vertical and attached to the slope even if the angle of inclination of the slope is changed.

4.2 Empirical Evaluation

To ensure that design decisions concerning the user interface were not made on the basis of untested hypotheses, *formative evaluation*[22] was used. Formative evaluation is the approach of building prototypes and testing them on users to obtain feedback that can be used to *inform* the next stage of the iterative design process. The types of question that formative evaluation was used to answer are:

- Which possible design solution should be adopted?
- What are the problems (if any) with the chosen solution?

There are various methods of formative evaluation, some of which are based over a longer timescale than others. Due to the timescale of this project, it was felt that the most appropriate method for formative evaluation would be that of prototyping and informal user testing during the development of the system, followed by controlled evaluation of the final product. Informal user testing was conducted according to a six-stage framework[22]: see Appendix D.

4.3 Evaluation of Final System

To evaluate the usability of the completed system, a controlled experiment was conducted. The purpose of performing a controlled experiment was to obtain measures of usability levels with a greater level of accuracy than could be achieved through informal testing.

The hypothesis explored by the controlled experiment was:

"The system results in a significant improvement in usability over existing constraint-based CAD systems."

To test this hypothesis, a *two-sample* experiment was devised. A two-sample experiment involves testing the two systems in question, collecting performance data, and then comparing the sample means of the data.

4.3.1 Identification of Variables and Choice of Tasks

Prior to devising tasks to be performed, the usability factor being evaluated (the dependent variable) was identified as the speed of performance of each task. The systems being tested were considered to be the independent variables.

The tasks devised were chosen to be simple enough to be performed in a very short timescale, but complex enough to be a realistic representation of tasks that potential users of such a system might perform. Each user was asked to create five simple geometric figures using the geometric primitives and constraints provided by the system (see Figure 4.4).

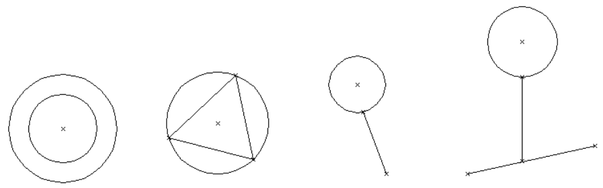


Figure 4.4: Geometric figures to be created. From left to right: concentric circles, triangle in circle, balloon, tree on slope, house.

To ensure that *nuisance variables* such as the description of the task given to the user, or the order in which the user was asked to create the geometric figures did not affect results, each user was asked to perform the set of tasks in exactly the same order, and a script was read to each user describing the task to be performed (see Appendix E).

4.3.2 Recruiting Subjects

My system was tested on a sample population of fifteen Cambridge University undergraduates. The students were chosen from a mixture of subjects. The results of these tests were compared with the time taken by an expert user to perform the tasks on an commercial constraint-based CAD system. It was hoped that my my system would be faster to use, even by novices, than an commercial constraint-based CAD system. Ideally, I would have liked to let the students attempt to use the commercial constraint based CAD system as well, however this proved to be infeasible due to financial and time constraints.

4.3.3 Collecting Data

Each of the subjects testing my system was read a script describing the system and given a demonstration of the creation of each type of geometric entity and constraint. The time taken by each subject to perform each of the tasks described in Section 4.3.1 was recorded. Prior to performing each task, each subject was read a description of the task to be performed and shown a picture of the geometric figure to be created.

This procedure was repeated so that two sets of results were obtained for each task.

4.3.4 Analysis of Data

The sample means of both sets of results obtained for my system were taken (see Table 4.1).

Task	Mean of First Set (seconds)	Mean of Second Set (seconds)
Concentric circles	12.8	9.0
Triangle in circle	69.8	31.0
Balloon	21.5	16.8
Tree on slope	34.8	22.5
House	52.3	28.5

Table 4.1: Mean times taken by an novice users using the system implemented.

These means were compared to the time taken by an expert user to perform the same tasks on an commercial constraint-based CAD system (see Table 4.2).

Task	Mean time (seconds)
Concentric circles	7.4
Triangle in circle	28.9
Balloon	7.1
Tree on slope	21.4
House	19.7

Table 4.2: Times taken by an expert using a commercial CAD system.

For each task, the time taken by the expert user on an commercial CAD system was faster (but not dramatically so) than the sample mean of both sets of data for my system.

To determine whether this difference between the sample mean and the time taken by an expert user on a different constraint-based CAD system is the result of normal random variation, or a usability difference, *confidence intervals* were calculated. This was done using a simplified form of the *t-test*[22]. Calculation of confidence intervals did not establish that the commercial constraint-based CAD system is more usable than my system – the difference between the sample means of each task for my system and time taken on the commercial system would appear to be from normal random variation.

While this test showed no clear usability advantage of one system over the other, commercial constraint-based CAD systems have a sufficiently high *abstraction barrier* that many architectural firms employ professional programmers to use the software. Since novice users managed to achieve comparable results using my system to those of an expert using an commercial constraint-based CAD system, I believe that my system would exhibit a clear usability advantage if both systems were tested using novice users.

4.4 Summary

This chapter described the testing and evaluation techniques employed in this project. In addition, examples of the the type of geometric figures that can be created using the completed system were included. Although it was not possible to prove a significant usability advantage over commercial constraint-based CAD systems, the project has been successful in meeting the requirements outlined in Section 2.1.1, as well as the overall aims stated in my project proposal.

5

Conclusion

The aim of this project was to design a constraint-based CAD system with a contraint representation that has a lower abstraction barrier than commercial constraint-based CAD packages. Such a package would have applications in the architectural and design industries, as was described in the Introduction chapter.

5.1 Achievements

A constraint-based CAD package satisfying the overall aim of the project, as well as the requirements of Section 2.1.1 was successfully designed, implemented, and subjected to usability testing.

A novel approach was taken to the design of the visual representation of constraints, by adopting a cross-disciplinary approach and using research on the display of layered information for air traffic control screens[26]. Constraints are represented as translucent infilled areas bounding the geometry they constrain.

The QuickPlan algorithm[34] was used as the basis of the constraint solver, enabling hierarchies of multi-way multi-output constraints to be satisfied with worst case time complexity of $O(C^2)$, where C is the number of constraints in the system. To my knowledge, this is a completely new approach, since the QuickPlan algorithm has not been used in any other constraint-based CAD systems to date.

To demonstrate the use of my constraint representation, as well as the power of the QuickPlan algorithm, a drawing package and several constraints were implemented, allowing geometric structures to be created and saved to disk.

Usability testing demonstrated that novice users can indeed use the system implemented, and achieve comparable results to an expert performing the same tasks on a commercial constraint-based CAD system. This is a significant step forward, since it is unlikely that novice users would be able to use such commercial systems without detailed training.

5.2 Future Development

Future work includes implementing a larger suite of constraints and a cyclic subsolver to enable constraint systems involving simultaneous linear equations to be solved. In addition, further research on constraint-based CAD systems from a point of view of end-user programming should be carried out. It would be of interest to compare the times taken by a sample of working architects to create simple geometric structures using my system with the times they take to perform the same tasks using commercial constraint-based CAD systems: my system is likely to provide a significant usability improvement.

Bibliography

[1] R. Aish. Custom objects: A model-oriented end-user programming environment. 2000.

[2] R.J. Anderson. Software engineering i, 2001.

[3] G.J. Badros. Constraints in interactive graphical applications, December 1998.

[4] A.F. Blackwell. Metaphor or analogy: How should we see programming abstractions? In P. Vanneste, K. Bertels, B. De Decker, and J.-M. Jaques, editors, *Proceedings of the $8^t h$ Annual Workshop of the Psychology of Programming Interest Group*, pages 105–113, April 1996.

[5] A.F. Blackwell. Human computer interaction, 2000-2001.

[6] A.F. Blackwell. Swyn: A visual representation for regular expressions. In H. Lieberman, editor, *Your Wish is My Command: Giving Users the Power to Instruct Their Software*, pages 245–270. Morgan Kauffman, 2001.

[7] A.F. Blackwell and T. Green. Does metaphor increase visual language usability? In *1999 IEEE Symposium on Visual Languages VL'99*, pages 246–253, 1999.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, twentieth printing edition, 1998.

[9] D. Flanagan. *Java in a Nutshell*. O'Reilly, second edition edition, 1997.

[10] D. Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly, 1999.

[11] D.M. Geary. *Graphic Java 2, Volume II: Swing*. Sun Microsystems Press: A Prentice Hall Title, 1999.

[12] M. Gleicher and A. Witkin. Drawing with constraints. *The Visual Computer*, 11(1):39–51, 1994.

[13] T. Green and A.F. Blackwell. Design for usability using cognitive dimensions. 1998.

[14] W3C XML Core Working Group. Extensible markup language (xml) 1.0 (second edition), 2000.

[15] A. Heydon and G. Nelson. The juno-2 constraint-based drawing editor. *Research Report 131a, Compaq Systems, Research Center, Palo Alto, CA*, December 1994.

[16] W. Hower and W. Graf. A bibliographical survey of constraint-based approaches to cad, graphics, layout visualization, and related topics. *Knowledge-Based Systems*, 9(7):449–464, December 1996.

[17] C.-Y. Hsu and B. Brüderlin. Constraint objects: Integrating constraint definition and graphical interaction. In *Proceedings on the Second Symposium on Solid Modeling and Applications*, pages 467–468, May 1993.

[18] J. Maloney, A. Borning, and B. Freeman-Benson. User-interface construction with constraints. In M. Burnett, A. Goldberg, and T.G. Lewis, editors, *Visual Object-Oriented Programming*. Manning Publications, 1995.

[19] B. McLaughlin. *Java and XML*. O'Reilly, 2000.

[20] W.J. Mitchell, R.S. Liggett, and T. Kvan. *The Art of Computer Graphics Programming: a Structured Introduction for Architects and Designers*. New York: Van Nostrand Reinhold, 1987.

[21] The Java Document Object Model. http://www.jdom.org.

[22] W. Newman and M. Lamming. *Interactive System Design*. Addison-Wesley, 2000.

[23] D.A. Norman. Cognitive artifacts. In John M. Carroll, editor, *Designing Interaction: Psychology at the Human-Computer Interface*, chapter 2, pages 17–38. Cambridge University Press, June 1991.

[24] S. Pantham. *Pure JFC 2D Graphics and Imaging*. Sams Publishing, 2000.

[25] M. Petre and A.F. Blackwell. Mental imagery in program design and visual programming. In *International Journal of Human-Computer Studies*, number 51(1), pages 7–30, 1999.

[26] L. Reynolds. Air traffic control screens: Choosing colour palettes for layered information. In *Information Graphics: Innovative Solutions in Contemporary Design*. Thames and Hudson, 1998.

[27] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 137–146, November 1994.

[28] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software – Practice and Experience*, 23(5):529–566, May 1983.

[29] B. Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Intelligent User Interfaces*, pages 33–39, 1997.

[30] S. Sistare. Graphical interaction techniques in constraint-based geometric modelling. In *Proceedings of Graphics Interface '91*, pages 85–92, Calgary, Alberta, June 1991.

[31] I. Sommerville. *Software Engineering (5th Edition)*. Addison-Wesley, 1995.

[32] B. Stroustrup. What is "object-oriented programming"?, 1991.

[33] X. Chen W. Bouma, I. Fudos, C. Hoffmann, and P.J. Vermeer. An electronic primer on geometric constraint solving.

[34] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

A

QuickPlan's Planning Phase

The planning stage of the algorithm is best described in graph-theoretic terms. Consider the constraint system to be an undirected graph, $G_C = (V, C, E)$, where the vertices of the graph, V and C, are sets of variables and constraints, and the edges, E, represent the relationship between the variables and constraints (*i.e.* if there is an edge between the vertex representing variable v and the vertex representing constraint c, then constraint c contains variable v). Assigning methods to constraints is equivalent to directing G_C such that a directed edge from a variable to a constraint represents the input to a method, and a directed edge from a constraint to a variable signifies that the selected constraint method determines the value of that variable. A constraint system is satisfiable if one method can be selected for each constraint resulting in a directed, acyclic constraint graph with no variable having more than one incoming edge. Such constraint graphs are known as *solution graphs* [34].

QuickPlan makes use of constraint hierarchies (see Section 2.2.3) to handle under-constrained systems. QuickPlan also assumes the existence of an implicit *stay constraint* [1] attached to every variable not determined by an explicit constraint. Therefore, every variable can always be considered to be determined by at least one constraint.

Constraint systems can have more than one valid solution graph. In such cases, QuickPlan identifies the "best" solution using the *locally-graph-better* comparator [34]. Consider the required constraints in a solution graph to be of level C_0, and the non-required constraints to be at levels C_1 through C_n. A solution graph is locally-graph-better than other solution graphs if, for a given constraint hierarchy, the solution graph satisfies all the constraints satisfied by the other solution graphs at levels C_0 through C_k, for some $k \leq n$, as well as at least one more constraint at level k. Occasionally, the locally-graph-better comparator will not identify one best solution. In this case, other techniques (such as stay constraints) are used.

The main principle behind the operation of the planning stage is propagation of degrees of freedom. For each constraint to be satisfied, the algorithm identifies the method belonging to that constraint with the smallest number of outputs, and searches for a set of *free variables* [2] consisting of those outputs. If such a set is found, this method is chosen to satisfy the constraint, and the constraint and its variables are removed from the constraint graph. If the algorithm reaches a point where no constraint has sufficient free variables to be assigned a method, QuickPlan does not immediately terminate, but instead retracts the weakest strength constraint from the subgraph (and saves it onto a priority queue) and then attempts to satisfy the new subgraph. This sequence of elimination and retraction steps is repeated until all constraints have been eliminated and a solution graph remains, or the algorithm encounters a cyclic subgraph of required constraints in which no constraint has sufficient free variables to be assigned a method.

[1] A stay constraint constrains a variable to remain unchanged.

[2] A free variable is a variable that is referenced by only one constraint.

If the constraint solver succeeds in enforcing the constraint but retracts constraints in the process, then the constraint solver attempts to enforce any retracted constraints since enforcing one or more of these retracted constraints would lead to a locally-graph-better solution. The set of potentially enforceable retracted constraints is restricted to constraints that satisfy the following criteria:

- The constraint must be of equal or lesser strength than the newly enforced constraint (if a higher strength constraint were to become enforceable then the previous solution would not have been a locally-graph better solution).
- The constraint must be downstream of the newly enforced constraint's variables (since the act of enforcing the newly enforced constraint may have affected downstream variables, thus allowing some downstream constraint to be enforced).
- The constraint must have a weaker upstream constraint (the weaker constraint can then be retracted, allowing other constraints to determine its variables, resulting in all downstream constraints of the weaker constraint becoming potentially enforceable).

To further improve efficiency, QuickPlan may annotate each variable with a *walkbound*, the strength of the weakest upstream constraint of the variable. If walkbounds are used, QuickPlan will only attempt to enforce a retracted constraint that has a method with an output variable of walkbound of strength less than its strength. This technique is not used for constraint graphs containing a small number of constraints.

B

QuickPlan Pseudo-Code

Pseudo-code for the QuickPlan algorithm is presented below. For more details regarding the algorithm, see [34].

`collectUpstreamConstraints` uses a depth first search to collect enforced constraints upstream of the constraint to be enforced:

```
collectUpstreamConstraints(Constraint cn)
    cn.mark = visitedMark

    // all upstream constraints whose strength is less than the strength
    // of the constraint to be enforced should be added to
    // retractableCnsQueue
    if (cn.strength < cnToEnforce.strength)
        retractableCnsQueue = retractableCnsQueue ∪ {cn}
    for each v ∈ cn.variables

        // compute v's numConstraints field
        if (v.mark == visitedMark)
            v.numConstraints = v.numConstraints+1;
        else
            v.mark = visitedMark
            v.numConstraints = 1
        e = v.determinedBy
        if ((e != null) && (e.mark != visitedMark))
            collectUpstreamConstraints(e)

        // input variables not being visited for the first time and
        // variables that have not yet been visited by any other constraint
        // than the constraint that outputs them are potential free variables
        else if (v.numConstraints == 1)
            freeVariableSet = freeVariableSet ∪ {v}
```

`collectDownstreamUnenforcedConstraints` collects unenforced constraints that are either attached to or downstream of `v`, and whose strength is less than or equal to the strongest constraint retracted to satisfy a newly enforced constraint:

```
collectDownstreamUnenforcedConstraints(Variable v)
    v.mark = searchMark
    unenforcedCnsQueue = unenforcedCnsQueue
            ∪ {cn | cn ∈ v.constraints, cn.selectedMethod == null,
           cn.strength ≤ strongestRetractedStrength}
    for each cn ∈ v.constraints
        if ((cn.selectedMethod != null) && (cn.mark != searchMark))
            cn.mark = searchMark
```

```
            for each w ∈ cn.selectedMethod.outputs
                if (w.mark != searchMark)
                    collectDownstreamUnenforcedConstraints(w)
```

Variable v's walkbound is the minimum of the strength of the constraints outputting v and the minimum strength constraint that would have to be revoked to enforce a method that does not currently output that variable:

```
computeWalkbound(Variable v)
   cn = v.determinedBy
   v.walkbound = cn.strength
   for each method ∈ {mt | mt ∈ cn.methods, v ∉ mt.outputs}
       maxWalkbound = max {w.walkbound | w ∈ (mt.outputs
               - cn.selectedMethod.outputs)}
       v.walkbound = min(v.walkbound, maxWalkbound)
```

`collectUnenforcedConstraints` collects unenforced constraints and prunes them using walkbounds if necessary:

```
collectUnenforcedConstraints(Set undeterminedVars, Constraint newlyEnforcedCn)
    for each v ∈ newlyEnforcedCn.selectedMethod.outputs ∪ undeterminedVars
        collectDownstreamUnenforcedConstraints(v)

    // if the total number of unenforced cosntraints exceeds
    // WALKBOUND_THRESHOLD the walkbounds are used to cull the set of
    // unenforced constraints.
    if (|unenforcedCnsQueue| > WALKBOUND_THRESHOLD)
        for each var ∈ newlyEnforcedCn.selectedMethod.outputs
            if (newlyEnforcedCn is an input or stay constraint)
                var.walkbound = newlyEnforcedCn.strength
            else
                var.walkbound = strongestRetractedStrength
        propagate walkbounds to variables downstream of variables in
                undeterminedVars ∪ newlyEnforcedCn.selectedMethod.outputs
        unenforcedCnsQueue = unenforcedCnsQueue -
                {cn | ∀ mt.outputs ∈ cn.methods ∃w ∈ mt such that
                w.walkbound ≥ cn.strength}
```

`multiOutputPlanner` propagates degrees of freedom to find acyclic solutions to sets of constraints:

```
multiOutputPlanner()
    while ((cnToEnforce.selectedMethod == null) && (freeVariableSet!= ∅))

        // remove an arbitrary element from the free variable set
        freeVar = removeElement(freeVariableSet)
        if (freeVar.numConstraints == 1)
            cn = the constraint to which freeVar belongs whose mark
                    equals visitedMark
            if (∃ mt ∈ cn.methods such that ∀ var ∈ mt.outputs,
                    var.numConstraints == 1)
                mt = the method withh the smallest outputs that satisfies
                        the above condition

                // any variable that is no longer output by cn and which
```

```
                // does not become a free variable is a potentially
                // undetermined variable
                for each var ∈ cn.selectedMethod.outputs - mt.outputs
                    var.determinedBy = null
                    if (var.numConstraints > 2)
                        potentialUndeterminedVars =
                                potentialUndeterminedVars ∪ {var}
                    else
                        var.mark = potentiallyUndetermined

                // keep track of redetermined constraints so they can be
                // undone if necessary
                undoStack.push(cn)
                undoStack.push(cn.selectedMethod)
                cn.selectedMethod = mt

                // the output variables' determinedBy fields must be set so
                // that collectUpstreamConstraints can perfrom its
                // reverse depth first search
                for each outputs ∈ mt.outputs
                    output.determinedBy = cn
                for each var ∈ cn.variables
                    var.numConstraints = varNumConstraints - 1
                    if (var.numConstraints == 1)
                        freeVariableSet = freeVariableSet ∪ {var}

                // a constraint can be removed from the set of
                // unsatisfied constraints by setting its mark field to null
                cn.mark = null

        // a variable that cannot be made the output of a constraint
        // and which is marked potentiallyUndetermined is a potentially
        // undetermined variable
        else if (freeVar.mark == potentiallyUndetermined)
            potentialUndeterminedVars = potentialUndeterminedVars ∪ {freeVar}
    else if (freeVar.mark == potentiallyUndetermined)
        potentialUndeterminedVars = potentialUndeterminedVars ∪ {freeVar}
```

`constraintHierarchyPlanner` is a propagate degrees of freedom algorithm that may retract constraints so that a higher strength constraint can be satisfied:

```
// ceiling strength is the maximum strength constraint that may be
// retracted in order to enforce a constraint
constraintHierarchyPlanner(int ceilingStrength)
    multiOutputPlanner()
    while ((cnToEnforce.selectedMethod == null) &&
            (retractableCnsQueue != null))
        cn = deleteMin(retractableCnsQueue)

        // indicate that the constraint has been removed from the set
        // of unsatisfied constraints by setting its mark field to null
        cn.mark = null
        strongestRetractedConstraintStrength =
                max(strongestRetractedConstraintStrength, cn.strength)

        // the outputs of the retracted constraint's selected method
        // become potentially undetermined variables
        for each output ∈ cn.selecteMethod.outputs
```

```
            output.determinedBy = null
            if (output.numConstraints > 2)
                potentialUndeterminedVars =
                        potentialUndeterminedVars ∪ {output}
            else
                output.mark = potentiallyUndetermined

        // keep tract of redetermined constraints so they can be
        // undone if necessary
        undoStack.push(cn)
        undoStack.push(cn.selectedMethod)
        cn.selectedMethod = null
        for each ∈ cn.variables
            v.numConstraints = v.numConstraints - 1
            if (v.numConstraints = 1)
                freeVariableSet = freeVariableSet ∪ {v}
        multiOutputPlanner()
```

`constraintHierarchySolver` attempts to find a locally-graph-better solution:

```
constraintHierachySolver()
    while (UnenforcedCnsQueue == ∅)
        cnToEnforce = deleteMax(unenforcedCnsQueue)
        visitedMark = generateUniqueMark()
        searchMark = generateUniqueMark()
        strongestRetractedConstraint = WEAKEST_CONSTRAINT_STRENGTH
        potentialUndeterminedVars = ∅
        retractableCnsQueue = ∅
        undoStack = ∅[1]
        collectUpstreamConstraints(cnToEnforce)

        // remove from free variable set any variables belonging to
        // more that one constraint
        freeVariableSet = freeVariableSet - {v | v.numConstraints > 1}
        constraintHierarchyPlanner(cnToEnforce.strength)

        // if the constraint could not be enforced...
        if (cnToEnforce.selectedMethod == null)

            // undo alterations to constraint graph
            while (undoStack != ∅)
                restoredMethod = undoStack.pop()
                cn = undoStack.pop()
                for each var ∈ cn.selectedMethod.outputs
                    var.determinedBy = null
                for each var ∈ restoredMethod.outputs
                    var.determinedBy = cn
                cn.method = restoredMethod

         // collect unenforced constraints only if a constraint was
         // retracted and the enforced constraint is not an input constraint
         else if ((strongestRetractedConstraint.strength >
                 WEAKEST_CONSTRAINT_STRENGTH) && (cnToEnforce is not an input
                 constraint that overrides a previously enforced stay constraint)

              // collect unenforced constraints downstream of either
```

[1] This line is not in the pseudo-code presented in [34], yet the algorithm does not function correctly without it.

```
            // the newly undetermined variables, or the outputs of the
            // newly enforced constraint
            undeterminedVariables = {w | w ∈ potentialUndeterminedVars,
                    w.determinedBy == null} ∪ { w | w ∈ freeVariableSet,
                    w.determinedBy == null, w.mark == potentiallyUndetermined}
            collectUnenforcedConstraints(undeterminedVariables, cnToEnforce)
```

addConstraint and removeConstraint add and remove constraints from the constraint system repectively:

```
addConstraint(Constraint cnToAdd)
    for each v ∈ cnToAdd.variables
        v.constraints = v.constraints ∪ {cnToAdd}
    if ∃ mt ∈ cnToAdd.methods such that for each v ∈ mt.outputs,
            |v.constraints| = 1
        cnToAdd.selectedMethod = mt
        for each v ∈ mt.outputs
            v.determinedBy = cnToAdd
    else
        unenforcedCnsQueue = {cnToAdd}
        constraintHierarchySolver()
        if ((cnToAdd.selectedMethod == null) && (cnToAdd.strength == required))
            print(constraint could not be added)

removeConstraint(Constraint cnToRemove)
    unenforcedCnsQueue = ∅
    strongestRetractedStrength = cnToRemove.strength
    for each v ∈ cnToRemove.variables
        v.constraints = v.constraints - {cnToRemove}
        if (v.determinedBy == cnToRemove)
            collectDownstreamUnenforcedConstraints(cnToRemove)
            v.determinedBy = null[2]
    constraintHierarchySolver()
```

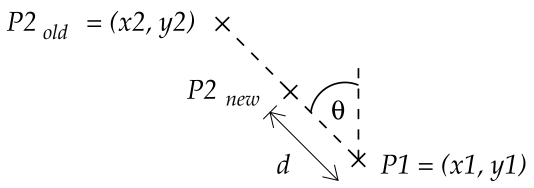
[2]This line is not in the pseudo-code presented in [34], yet the algorithm does not function correctly without it.

C

Constraints

C.1 Distance Between Two Points

The `DistanceConstraint` class constrains a `DoublePoint2D`, $P1$, to be a certain distance from another `DoublePoint2D`, $P2$, and vice versa. The angle of inclination of the line between $P1$ and $P2$ remains unchanged when the constraint is executed (see Figure C.1).

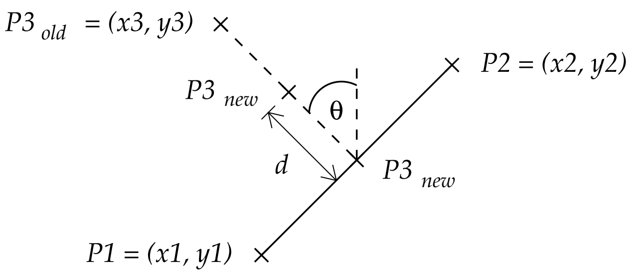


$$P2_{new} = (x1 + d\ \sin\theta,\ y1 + d\ \cos\theta)$$

Figure C.1: Constraining $P2$ to be a distance d from $P1$.

C.2 Distance Between a Point and Line

The `PointLineDistanceConstraint` class constrains a `DoublePoint2D`,$P1$, to be a certain distance from a `DoubleLine2D` (see Figure C.2).



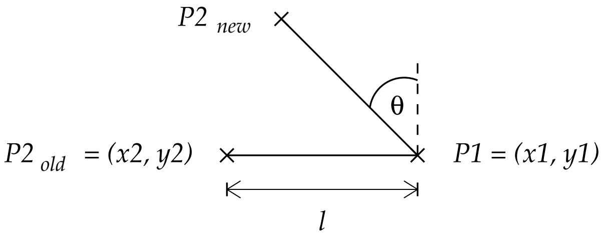
$$P3_{new} = \begin{cases} (x1 + t\,(x2 - x1) + d\ \sin\theta,\ y1 + t\,(y2 - t1) + d\ \cos\theta) & : \ 0 < t < 1 \\ (x1 + d\ \sin\theta,\ y1 + d\ \cos\theta) & : \ t \leq 0 \\ (x2 + d\ \sin\theta,\ y2 + d\ \cos\theta) & : \ t \geq 1 \end{cases}$$

$$\text{where } t = \frac{(x3 - x1)(x1 - x2) + (y3 - y1)(y1 - y2)}{(x2 - x1)(x1 - x2) + (y2 - y1)(y1 - y2)}$$

Figure C.2: Constraining $P3$ to be a distance d from the line segment between $P1$ and $P2$.

C.3 Angle of Inclination of a Line

The `LineAngleConstraint` class constraints the angle of inclination of a `Double-Line2D`. The length of the `DoubleLine2D` is kept constant and the location of one or other of its `DoublePoint2D` end-points is changed (see Figure C.3).

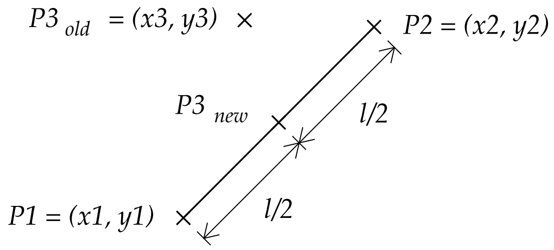


$$P2_{new} = (x1 + l\ \sin\theta,\ y1 + l\ \cos theta)$$

Figure C.3: Constraining the angle of inclination of the line segment between $P1$ and $P2$ to be θ.

C.4 Point on Mid-Point of a Line Segment

The `MidPointLineConstraint` class constrains a `DoublePoint2D` to lie at the mid-point of a `DoubleLine2D` (see Figure C.4).

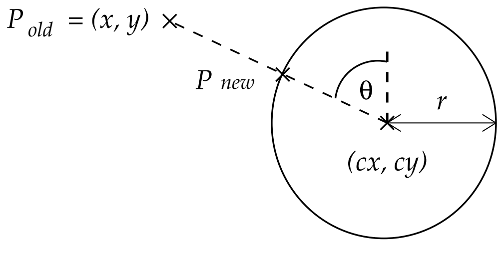


$$P3_{new} = (x1 + \frac{(x2\ -\ x1)}{2},\ y1 + \frac{(y2\ -\ t1)}{2})$$

Figure C.4: Constraining $P3$ to lie at the mid-point of the line segment between $P1$ and $P2$.

C.5 Point on the Circumference of Circle

The `PointOnCircleConstraint` class constrains a `DoublePoint2D` to lie anywhere on a `DoubleCircle2D` (see Figure C.5).

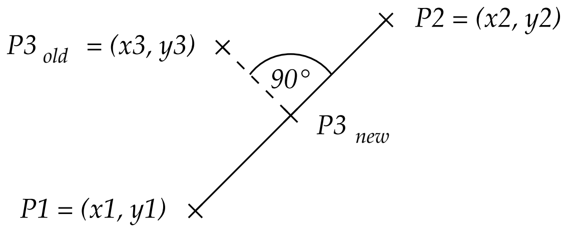


$$P3_{new} = (cx + r\ \ \sin\theta,\ cy + r\ \ \cos theta)$$

Figure C.5: Constraining a point, P, to lie on a circle of radius r.

C.6 Point on a Line Segment

The `PointOnLineConstraint` class constrains a `DoublePoint2D` to lie anywhere on a `DoubleLine2D` (see Figure C.6).



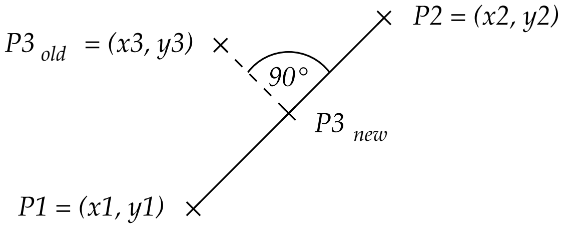
$$P3_{new} = \begin{cases} (x1 + t\,(x2 - x1),\ y1 + t\,(y2 - t1)) & : \ 0 < t < 1 \\ (x1,\ y1) & : \ t \leq 0 \\ (x2,\ y2) & : \ t \geq 1 \end{cases}$$

$$\text{where } t = \frac{(x3 - x1)(x1 - x2) + (y3 - y1)(y1 - y2)}{(x2 - x1)(x1 - x2) + (y2 - y1)(y1 - y2)}$$

Figure C.6: Constraining a point, $P3$, to lie on the line segment between $P1$ and $P2$.

C.7 Point on a Line Segment Extended

The `PointOnLineExtendedConstraint` class constrains a `DoublePoint2D` to lie anywhere on the extension of a `DoubleLine2D` (see Figure C.7).



$$P3_{new} = (x1 + t\,(x2\ -\ x1),\ y1 + t\,(y2\ -\ t1))$$

where $t = \frac{(x3\ -\ x1)(x1\ -\ x2)\ +\ (y3\ -\ y1)(y1\ -\ y2)}{(x2\ -\ x1)(x1\ -\ x2)\ +\ (y2\ -\ y1)(y1\ -\ y2)}$

Figure C.7: Constraining a point, $P3$, to lie on the line segment between $P1$ and $P2$ extended.

D

Informal User Testing

Throughout the development process, informal user testing was used to determine whether design decisions made were appropriate and would facilitate solution of the overall design problem.

- Usability properties to be evaluated were identified.
- Prototypes were designed to have sufficient functionality to allow the user to perform tasks relating to the usability properties of interest. Java was used as the prototyping tool, enabling prototypes to be refined into the final product.
- Experiments were devised to test the usability properties of interest as fully as possible. The users to participate in the testing were selected.
- Direct observation and *think-aloud* studies were used to collect data while users were performing the tasks devised to test usability.
- Analysis of data obtained was performed, to identify of good and bad features of the design.
- Conclusions were drawn about which aspects of the design needed changing and the nature of these changes. Aspects of the design that were identified as successful were recorded, so they could be retained in subsequent revisions of the prototype.

This six-stage framework was used to evaluate both the drawing package and visual representation of constraints, as well as the system as a whole.

E

Scripts Used During Experimental Evaluation

The script read to subjects testing my system to provide an overview of my constraint-based CAD system is below:

> *This is a constraint-based drawing package. You may create points, line segments and circles. Points can be created by clicking the mouse in the desired location. Line segments and circles can be created by pressing the mouse and dragging the mouse. Points, line segments and circles may all be constrained.*
>
> *You can:*
>
> - *Fix a point's location.*
> - *Fix the distance between two points.*
> - *Fix the distance between a point and a line segment.*
> - *Constrain two points to be coincident.*
> - *Constrain a point to lie at the mid-point of a line segment.*
> - *Constrain a point to lie on a line segment.*
> - *Constrain a point to lie on a line segment extended.*
> - *Constrain a point to lie on the circumference of a circle.*
> - *Constrain the angle of inclination of an angle.*
> - *Constrain a line to be horizontal.*
> - *Constrain a line to be vertical.*
>
> *To create a constraint, select the appropriate constraint creation tool from the constraint toolbar and then click on the geometric entity or entities to be constrained.*

The scripts read to each subject testing my system and to the expert performing the same tasks on a commercial constraint-based CAD system prior to undertaking each task:

> *Task 1: I would like you to draw two concentric circles.*

> *Task 2: I would like you to draw a triangle within a circle. The vertices of the triangle should lie on the circumference of the circle. You should create the triangle first and then constrain the vertices of the triangle to lie on the circumference of the circle.*

> *Task 3: I would like you to draw a balloon. The string of the balloon should be a fixed length, and should remain in contact with the balloon at all times.*

Task 4: I would like you to draw a tree on a slope. The tree trunk should be vertical at all times. The tree trunk should be joined onto the slope at all times. The tree trunk should be joined onto the leafy part of the tree at all times.

Task 5: I would like you to draw a house with a pitched roof. The floor of the house should be horizontal, and the walls vertical. The walls should remain in contact with the floor at all times. The roof should remain in contact with the walls at all times.

F

Sample Code

F.1 QuickPlan's Execution Phase

After a constraint is removed or a new constraint is added, the constraints in the system are topologically sorted using the following methods:

```
private Vector topologicallySortConstraints() {
    int[] color = new int[constraints.size()];
    Vector sortedVertices = new Vector();

    for (int i=0; i<constraints.size(); i++) {
        Constraint cn = (Constraint) constraints.elementAt(i);
        color[i] = WHITE;
    }

    for (int i=0; i<constraints.size(); i++) {
        if (color[i] == WHITE)
            depthFirstSearchVisit(i, color, sortedVertices);
    }

    return sortedVertices;
}

private void depthFirstSearchVisit(int index, int[] color, Vector
sortedVertices) {
    color[index] = GREY;

    Constraint cn = (Constraint) constraints.elementAt(index);

    if (cn.getSelectedMethod() != null) {
        for (int i=0; i<cn.getSelectedMethod().outputs.length; i++) {
            DoublePoint2D v = cn.getSelectedMethod().outputs[i];

            for (int j=0; j<v.getConstraints().size(); j++) {
                int newIndex = constraints.indexOf(v.getConstraints().elementAt(j));
                if (color[newIndex] == WHITE) {
                    depthFirstSearchVisit(newIndex, color, sortedVertices);
                }
            }
        }
```

```
        color[index] = BLACK;
        sortedVertices.addElement(new Integer(index));
    }
}
```

To enforce the constraints in the system, the `enforce` method of each constraint must be invoked in topological order:

```
public void enforceConstraints(Vector sortedVertices) {
    for (int i=sortedVertices.size()-1; i>=0; i--)
        ((Constraint) sortedVertices.elementAt(i)).enforce();
}
```

F.2 MidPointLineConstraint

The `MidPointLineConstraint` class is shown below:

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

import java.lang.Math;

public class MidPointLineConstraint extends Constraint {
    private Color midPtLineConstraintColor = new Color(211, 159, 221);
    private DoubleLine2D line;

    public MidPointLineConstraint(DoublePoint2D point,
                                  DoubleLine2D line, int strength) {

        variables = new DoublePoint2D[1];
        methods = new Method[1];

        variables[0] = point;
        variables[0].setNumConstraints(variables[0].getNumConstraints()+1);

        this.line = line;
        this.methods[0] = new Method1();

        if (variables[0].isPartOfObject() &&
            !variables[0].getObjectIsPartOf().isAuxiliary()) {

            isAuxiliary = false;
        }

        else {
            isAuxiliary = true;
        }

        stay = false;
```

```
        inputFlag = true;

        setSelectedMethod(null);
        setStrength(strength);
    }

    public class Method1 extends Method {
        private double x1, x2, y1, y2, x, y;

        public Method1() {
            outputs = new DoublePoint2D[1];
            outputs[0] = variables[0];
        }

        public void execute() {
            x1 = ((DoublePoint2D) line.getP1()).x;
            y1 = ((DoublePoint2D) line.getP1()).y;
            x2 = ((DoublePoint2D) line.getP2()).x;
            y2 = ((DoublePoint2D) line.getP2()).y;

            x = x1 + (x2-x1)/2;
            y = y1 + (y2-y1)/2;

            variables[0].x = x;
            variables[0].y = y;
        }
    }

    public DoubleLine2D getLine() {
        return this.line;
    }

    public void drawConstraint(Graphics2D g2D, Drawing drawing) {

        // buffered image to contain constraint representation
        BufferedImage cnImage = new BufferedImage(40, 40,
                                          BufferedImage.TYPE_INT_ARGB);

        // main shape of constraint representation
        Area outerArea = new Area(new Ellipse2D.Double(variables[0].x-18,
                                               variables[0].y-18,
                                               36, 36));

        // constraint fill
        Area innerArea = new Area(new Rectangle2D.Double(variables[0].x-7,
                                                 variables[0].y-7,
                                                 14, 14));

        Graphics2D cnImageGraphics2D = cnImage.createGraphics();

        // translate it to the correct location for rendering on the
        // Drawing's Graphics2D context
        cnImageGraphics2D.translate(-(variables[0].x-18)+2,
                                    -(variables[0].y-18)+2);

        cnImageGraphics2D.setColor(midPtLineConstraintColor);
        cnImageGraphics2D.fill(outerArea);
        cnImageGraphics2D.setPaint(Color.white);
```

```
        cnImageGraphics2D.fill(innerArea);

        // create a new BufferedImage to be used as the destination
        // BufferedImage in the convolution operation.
        BufferedImage destImage = new BufferedImage(40, 40,
                                            BufferedImage.TYPE_INT_ARGB);

        // values to be used in filter kernel
        float[] data = {0.0625f, 0.125f, 0.0625f,
                        0.125f, 0.25f, 0.125f,
                        0.0625f, 0.125f, 0.0625f };

        // create filter kernel
        Kernel kernel = new Kernel(3, 3, data);

        // perform convolution
        ConvolveOp convolve = new ConvolveOp(kernel);
        convolve.filter(cnImage, destImage);

        g2D.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                            (float) 0.45));

        g2D.drawImage(destImage, (int) variables[0].x-20,
                      (int) variables[0].y-20, drawing);

        g2D.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                            (float) 1.0));
    }
}
```