Automatic Furniture Population of Large Architectural Models

by

Kari Anne Høier Kjølaas

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2000

© Kari Anne Høier Kjølaas. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author . . . .
Department of Electrical Engineering and Computer Science
January 30, 2000

Certified by . . . .
Seth Teller
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Automatic Furniture Population of

Large Architectural Models

by

Kari Anne Høier Kjølaas

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The goal of this project is to automate the process of generating large furnished models of building interiors. The FurnIt system was designed to automatically place furniture into a given floor plan. It deals solely with the spatial placement of furniture, ignoring esthetic concerns such as colors, fabrics and lighting.

FurnIt represents rooms, furniture groups and individual pieces of furniture as a nested hierarchy of spaces with assigned functions. The space of a piece of furniture is defined by its bounding box. Each level of the spatial hierarchy is represented by a template that specifies the configuration of all sub-spaces enclosed in the given space. A set of default templates are provided for most common room and furniture types.

FurnIt selects a template and adapts it to a given room, recursively resolving spatial conflicts in the room template's sub-spaces until a valid furniture configuration is reached. The result is guaranteed to be geometrically valid and functional.

FurnIt was developed as part of the City Walk project. It works in concert with other City Walk tools that generate textured building exteriors and interiors.

Thesis Supervisor: Seth Teller
Title: Associate Professor

Acknowledgments

First, I would like to thank Professor Seth Teller for supporting this project. I would also like to thank the other members of the City Walk group at MIT and UC Berkeley, especially Laura Downs. This system is built on top of her SYLIF library, and I have greatly benefited from her insights.

Thanks also go to Jane Gould and the people of the Lutheran-Episcopal Ministry at MIT. They tried to keep me sane throughout my MIT experience, never acknowledging the futility of their efforts.

Further thanks to Brian, who has had to deal with more cats, laundry and late minute cancellations than any man should have to.

Special thanks go to my parents Jorun and Per Oskar Kjølaas. Without their emotional support and deep wallets my MIT education would have been short-lived.

Finally, I would like to dedicate this work to the memory of my cousin, Audun Ryeng.

Come to me all you that are weary and carrying heavy burdens,
and I will give you rest. Mt 11.28

Contents

1 Introduction **9**

1.1 Background . . . 9

1.2 Goal . . . 10

1.3 Related Work . . . 11

1.4 Organization of Thesis . . . 11

2 Overview of Design **13**

2.1 Design Requirements . . . 13

2.1.1 Geometrical Validity . . . 13

2.1.2 Functional Validity . . . 13

2.1.3 Overall Room Layout . . . 14

2.2 Reasoning about Hierarchical Spaces . . . 15

2.3 Configuration Templates . . . 16

3 The FurnIt System **17**

3.1 Definition of Templates . . . 17

3.2 Initializing Templates . . . 18

3.2.1 Fitting templates to rooms . . . 21

3.2.2 Doors and Windows . . . 22

3.3 Resolving Conflicts . . . 23

3.3.1 Moving Child Templates . . . 24

3.3.2 Re-sizing Child Templates . . . 27

3.3.3 Recursion . . . 27

3.3.4 Base Cases . . . 28

4 Implementation **30**

4.1 FurnIt Control Flow 30

4.2 FurnIt Debugging Interface 31

4.3 SYLIF (SYmbolic Layout Interchange Format) 31

4.3.1 Overview of SYLIF Data Structures 33

4.3.2 SYLIF-FurnIt Interaction 34

4.3.3 SYLIF Template Classes 34

4.4 Floorsketch 39

4.4.1 Extensions 41

5 Results **43**

5.1 Using FurnIt 43

5.1.1 Creating Rooms 43

5.1.2 Adding Portals 44

5.1.3 Furnishing a Floor Plan 44

5.1.4 Manually Adjusting Furniture Configuration 45

5.2 Default Templates 45

5.2.1 Living Room 46

5.2.2 Dining Room 47

5.2.3 Master Bedroom and Bedroom 47

5.2.4 Office and Student Office 47

5.3 Variations in Furniture Configurations 49

6 Discussion and Conclusion **56**

6.1 Future Work 56

6.2 Conclusion 57

A Tables **58**

A.1 Default Room Template Types 58

A.2 Default Furniture Group Template Types 59

A.3 Default Furniture Template Types 60

A.4 Default Furniture Icons 61

B Setting Up and Running FurnIt **62**

B.1 Setting Up FurnIt 62

B.2 Running FurnIt 63

B.3 FurnIt Source Files 63

List of Figures

1-1 Hierarchy of patterns. Adapted from Alexander [2]. 12

3-1 Diagram of office template. 19
3-2 Possible mutations of room template. 20
3-3 Scaling of room template. 22
3-4 Adding "clear space" to a room template. 23
3-5 Resolving root template by moving children. 26
3-6 Re-sizing child template. 27

4-1 Control flow in FurnIt. 32
4-2 SYLIF class inheritance hierarchy. Adapted from Downs [8]. 36
4-3 Member variables of SYLIF template classes. 37
4-4 Floorsketch view of second floor of building NE43. 40
4-5 Interactions between City Walk applications. 42

5-1 Default living room template. 46
5-2 Default dining room template. 47
5-3 Default master bedroom template. 48
5-4 Default bedroom template. 48
5-5 Default faculty office template. 49
5-6 Default student office template. 50
5-7 First sample configuration. 51
5-8 Second sample configuration. 52
5-9 Third sample configuration. 53
5-10 Fourth sample configuration. 54
5-11 Fifth sample configuration. 55

List of Tables

A.1 Table of default room template types. 58
A.2 Table of default furniture group template types. 59
A.3 Table of default furniture template types. 60
A.4 Table of furniture icon colors. 61

Chapter 1

Introduction

1.1 Background

FurnIt is a system that automatically places furniture into a given floor plan. It was created as part of the City Walk project, a joint research project of MIT and UC Berkeley. The goal of City Walk is to integrate projects at the two universities in order to create a comprehensive system for generating and interactively walking through extended interior and exterior models of urban scenes.

The MIT City Scanning Project [17] is an effort to create CAD models of existing urban scenes using only images annotated with camera position and direction. It is headed by Prof. Seth Teller of the MIT Graphics Group. The goal of the project is to create a fully automated system that acquires the necessary images and processes these to produce valid models that will include geometric and texture information for all buildings in the scene. Currently the project is creating a model of the MIT campus, but the eventual goal is to develop a system that can handle the modeling of thousands of buildings.

The UC Berkeley Fire Walk project [4] is headed by Carlo Séquin. Over a period of years his group has developed a suite of tools for exploring virtual environments of both existing and planned buildings. Their WALKTHRU system [18] allows multiple users to walk through a large furnished building model in real-time. This building model is not static. Furniture can be placed and moved interactively with the WALKEDIT system [3], and physics and fire simulators that interact with the building model have been developed.

The goal of City Walk is to coordinate the efforts of these two projects, and to compile a suite of tools that enable a user to create models of large urban scenes and interact with

these, moving seamlessly between indoor and outdoor environments. In order to achieve this, data generated by the City Walk system must be automatically incorporated into the WALKTHRU model and integrated with data for building interiors. Furthermore, the generation of furnished building interiors from floor plans must be automated so that it can handle scenes containing large numbers of buildings. Finally, algorithms for large changes in object and viewing scale and multiple levels of detail must be developed.

1.2 Goal

FurnIt was designed to help automate the process of generating large furnished models of building interiors. Provided with a floor plan with room labels and wall, window and door information, we wish to create a model populated with furniture in a functional configuration. We have chosen a conservative approach, one guaranteeing a furniture configuration that is geometrically valid, if not always ideal. FurnIt is not guaranteed to find an optimal furniture configuration. It does guarantee a solution with no geometrical inconsistencies, i.e. no interpenetrating furniture, no furniture penetrating the walls, floor or ceiling and no furniture floating in the air.

The FurnIt system is fully automated when provided with a floor plan and a set of furniture templates. Automation is necessary since the primary use of the system will be for scenes too large to populate with furniture by hand. The goal of the system is not to do a better job than the average human user; rather it is to do a decent job in much less time. It was created to be used on very large models, where the time required to place furniture by hand would be prohibitive.

Note that FurnIt is intended as a system for furniture *placement*. It does not attempt to determine colors, fabrics, lighting or many of the other aspects of furnishing a building; but rather deals solely with the spatial aspect of selecting and placing furniture into a given set of rooms.

FurnIt was designed to work in concert with the Floorsketch system (see Section 4.4 on page 39), a program for interactively drawing 2D floor plans and automatically extruding these to 3D. Floorsketch is limited to rectangular rooms, and FurnIt has adopted this restriction. This ensures that the two applications are compatible. Furthermore, it greatly reduces the complexity of FurnIt by ensuring that we only have to reason about rectangular

spaces.

1.3 Related Work

Most of the previous work related to this topic has been carried out in the areas of artificial intelligence and architecture. These projects have been either much larger in scope (e.g. attempts at encoding basic architectural principles) or much narrower in focus (e.g. generating new designs that imitate the style of a given architect).

In their book *A Pattern Language* [2] Alexander et al. present "patterns" for towns, neighborhoods, houses and rooms. These patterns attempt to encode architectural configurations that are universal for a given space. Each pattern is presented using images and text. Examples of given room patterns are "window place", "workspace enclosure" and "cooking layout".

Although the authors try to formulate a phenomenology of architecture, their approach is not rigorous, and the presented patterns are not intended as the basis of a computational system. The work's strength lies in the authors' efforts to specify a hierarchy of patterns. Each pattern has a list of parent patterns and one of child patterns. Parent patterns represent spaces enclosing the space represented by the given pattern; child patterns represent spaces enclosed within it. The result is a hierarchy of spaces, recursively specifying the components of a given space. The graph in Figure 1-1 on the next page illustrates this. The pattern "workspace enclosure" is a child of the patterns " half-private office" and "home workshop". Its children include "half-open wall", "open shelves" and "sitting circle".

1.4 Organization of Thesis

This thesis describes the design and implementation of the FurnIt system. The next chapter gives an overview of the design criteria for a furniture placement system. It also contains a high level discussion of the FurnIt model. Chapter 3 presents a detailed description of the FurnIt data structures, supplemented with concrete examples. Chapter 4 details the implementation of the system and how it interacts with other City Walk applications. Finally, a wide range of results are presented, as well as a discussion of the system and areas of future improvements.

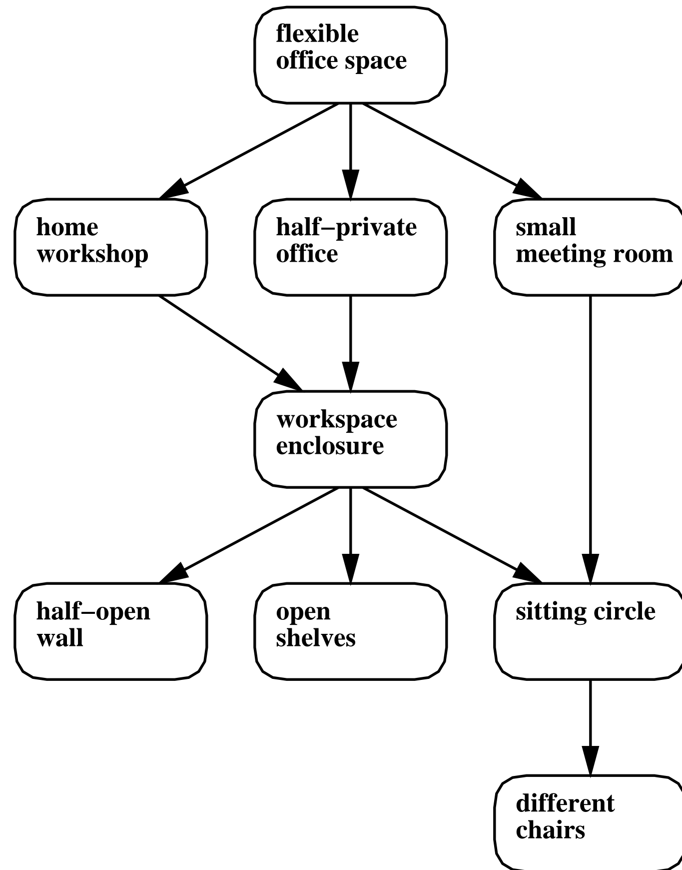


Figure 1-1: Hierarchy of patterns. Adapted from Alexander [2].

Chapter 2

Overview of Design

2.1 Design Requirements

Many different methods can be used to automatically determine a reasonable furniture configuration for a given floor plan. In order to choose the best method, it is helpful to consider the design requirements for such a system.

2.1.1 Geometrical Validity

As stated in our goal, the FurnIt system should guarantee geometrically valid output. All pieces of furniture should be contained within a room and not penetrate the walls, floor or ceiling. No two pieces of furniture should interpenetrate, and no piece of furniture should float in space.

It is reasonable to require each piece of furniture to be wholly contained within a single room. While it is possible for a piece of furniture to be in two rooms simultaneously without interpenetrating a wall (e.g. being placed in a doorway or other opening), this is not common and greatly increases the complexity of the representation.

2.1.2 Functional Validity

In order for FurnIt to be useful, the program must place furniture in a configuration that is functional. Furniture placed in a given room should be appropriate for the room type. Furthermore, a piece of furniture should not be placed without also placing other pieces of furniture necessary for its use. A dining room table is used for eating, which is usually done

while sitting down. Thus a dining room table should not be placed without its accompanying chairs.

Geometrical constraints can be stated and checked in a straightforward manner, although this can be computationally intensive. Functional constraints, on the other hand, can often be ambiguous. There are no clear rules to indicate what furniture should be placed in a given room. While some choices are clearly incorrect (one does not place a bath tub in an office), many other choices are ambiguous (in most cases an office will not contain a sofa, but sometimes this will be appropriate). Likewise, the interdependency of different pieces of furniture is not always clear. A living room sofa is almost always used with a coffee table, but such a configuration is not inevitable.

Thus a furniture placement system must take a conservative approach, allowing only configurations that are guaranteed to be valid. Instead of trying to define which configurations are illegal, it must instead define some criteria that guarantee legal configurations and accept only configurations that satisfy these. This approach will eliminate many acceptable configurations, but will guarantee to produce only functionally valid results.

2.1.3 Overall Room Layout

FurnIt places functional furniture groups only. It is not sufficient, however, merely to ensure that appropriate groups of furniture are placed in each room. The overall room layout must also be considered in order to create usable configurations. This is by far the most ambiguous of the requirements, and includes esthetic as well as functional aspects. Considerations such as symmetry, room composition etc. are highly subjective and will be resolved differently by different schools of design.

In our goal statement we specified that FurnIt should place furniture in a *reasonable* configuration. This limitation frees us from having to take into account a number of esthetic concerns, and we are able to consider a wide range of solutions. However, we can not completely ignore room layout. A balance of furniture and open space must be maintained and there must be open paths for the user to move between furniture groups.

2.2 Reasoning about Hierarchical Spaces

Our problem statement limits the system to resolving only the spatial aspect of the task of selecting and placing furniture in a given set of rooms. The system's design requirements further emphasize the spatial nature of the task.

Rooms can be represented as spaces with an assigned function and a given set of characteristics. Individual pieces and groups of furniture can likewise be considered spaces with assigned functions. A dining room is a space dedicated to meals. As a consequence, it encloses other spaces such as a dining room set and storage spaces for china and silverware. Similarly, a file cabinet lies within a space designated as storage, a chair within a space reserved for sitting. While the geometric shape of furniture can be complex, the shape of a given piece can be approximated by a tightly fitting, rectangular, bounding box around the furniture geometry. The bounding box approximation and our decision to limit the system to rectangular rooms ensures that we only need to reason about rectangular spaces.

One other insight gained from the design requirements is that furniture is organized in functional groups. Different types of furniture often work together to fulfill a given function, such as a meal. These pieces form an interdependent group, and any functional configuration must include all the members of such a group or none of them.

Note that the configuration of the furniture within a given group need not be determined until after the overall group is placed. First, one decides where to place the reading space within a living room; then one decides how to place the bookshelves, chair, side-table and other individual pieces of furniture within the group.

One last insight gained from the design requirements is that open space is necessary for room function. A desk needs not only to be accompanied by a chair, it also needs to be placed so that there is sufficient room to push the chair back from the desk. In general, we need a mechanism that ensures that individual pieces of furniture are surrounded by sufficient open space.

All of the above observations support an approach similar to that of Alexander [2]. We choose to express a given room as a nested hierarchy of spaces, each with its own characteristics. Each space can include one or more sub-spaces (child spaces), and is governed by rules for how these should be arranged. By recursively resolving spaces starting at a given room (root space), we can create a reasonable furniture configuration for that room.

2.3 Configuration Templates

The above observations define our overall framework for furniture population. However, the key to resolving spatial problems depends on our ability to encode the rules for arranging subspaces within a given space. We need a hierarchy of spaces similar to the hierarchy of patterns presented in Alexander [2]. In addition to such a hierarchy, we need a comprehensive set of rules for where to place child spaces within a parent space. These rules are both complex and ambiguous, as was shown in the discussion of the design requirements.

Faced with these difficulties, we have chosen a case-based approach. We assume that a set of reasonable configurations of child spaces is available. In FurnIt, these are created by hand by the user, and are expressed in the form of templates that give the location and orientation of each child space. Both furniture and templates are represented as rectangular solids. This makes it possible to allow templates to contain other templates, as well as furniture, efficiently encoding the spatial hierarchy (see Figure 3-1 on page 19).

Furthermore, we provide a library of mutations. These are operations that can be performed on templates, changing the location and orientation of child spaces within a template, as well as adding and removing elements from the template. No mutation should introduce geometrical or functional inconsistencies. The purpose of the mutations is to resolve any spatial conflicts, as well as to adapt a template to a room shape in such a way that many different furniture configurations can be obtained from a single template.

Requiring hand-made templates seems like a direct contradiction of our goal to create an automated system. Note, however, that only one template must be created for each room type. This template yields a range of furniture configurations. Furthermore, in the final system we have provided the user with a set of default templates, making it possible to furnish a range of common room types without having to create new templates.

We can now outline a simple algorithm for furnishing a room. First we pick a configuration template. We then adapt the template to the given room, fitting its space tightly inside the room and resolving any resulting inconsistencies by mutating the template until it is geometrically and functionally valid. Once the top level template is resolved, we recursively solve each of its child templates until we reach the bounding boxes of individual pieces of furniture. The next chapter discusses this process in detail.

Chapter 3

The FurnIt System

The previous chapter gave a high-level outline of the design of FurnIt. This chapter presents a more detailed description of the system, specifying the different data structures and giving a detailed algorithm for how these are used. Concrete examples are given.

3.1 Definition of Templates

A template specifies a configuration of subspaces within a given space. It is defined by its size as well as its position and orientation in a parent space. Each template has a type such as "workspace", "seating" or "office" that signifies its function in the parent space.

As previously discussed, we have limited FurnIt to reasoning about rectangular rooms and have chosen to represent pieces of furniture by their tight fitting axial bounding boxes. Thus any space represented in the FurnIt system has the shape of a rectangular solid, and its size can be specified by its width, depth and height. The position of a template can be expressed as the (x, y, z) coordinate of the template corner in the parent space. Note that the position of a template is constrained by the requirement that the template must be completely enclosed by its parent.

In most room types the majority of furniture is placed on the floor. Furthermore, furniture hung from a wall, e.g. a shelf or cupboard, is commonly oriented so that its lower edge is parallel to the floor plane. In this system we have chosen to limit the orientation of templates to a random rotation in the xy-plane. This limitation greatly reduces the complexity of spatial reasoning while leaving the system general enough to express all commonly found furniture configurations.

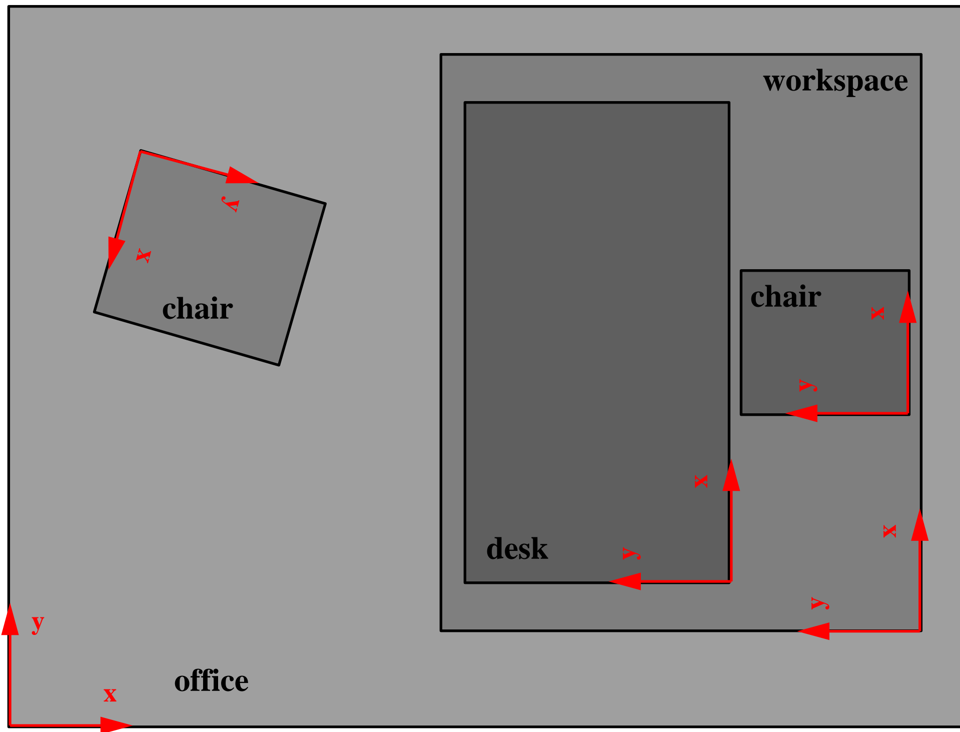
The purpose of a template is to encode the spatial configuration of its child templates. Each template contains a list of child templates whose locations and orientations are specified in the parent template's coordinate system. This configuration encodes the spatial relationship between the child templates. However, additional information regarding function is necessary. Not all child templates in a given space are equally important. In a bedroom, a bed is absolutely necessary for the room's function. A dresser, however, is both useful and common, but can be removed if necessary. We reflect this by allowing each child template to be flagged as vital or non-vital. This specifies the child's importance in the configuration of the parent template.

A room or a given piece of furniture has a range of functional sizes. Bedrooms commonly vary in size from $10' \times 10'$ to $16' \times 14'$; sofas can vary in length by several feet. We encode this by giving each template a minimum and maximum size, limiting its size to a specified range. This can be used to help to determine unresolvable templates after a minimum of computation (see Section 3.3 on page 23). Furthermore, minimum size can be used to ensure sufficient open space around a piece or group of furniture. Finally, minimum and maximum size can be used to specify relationships between the respective sizes of a template's children.

Figure 3-1 on the following page gives an example of a template that specifies a simple layout for an office. Note that the child template "workspace" has itself two child templates, "desk" and "chair". These are defined in the scope of the "workspace" template and contain no information regarding the overall room layout. The two "chair" templates do not lead to any conflicts since they are defined in different scopes.

3.2 Initializing Templates

The first step in populating a given room is to initialize a set of root templates that describes possible layouts of the room. This section describes the process of creating a set or root templates by mutating a given template and adding constraints due to the given room's doors and windows. These root templates often contain geometrical inconsistencies. Resolving such inconsistencies will be discussed in the next section.



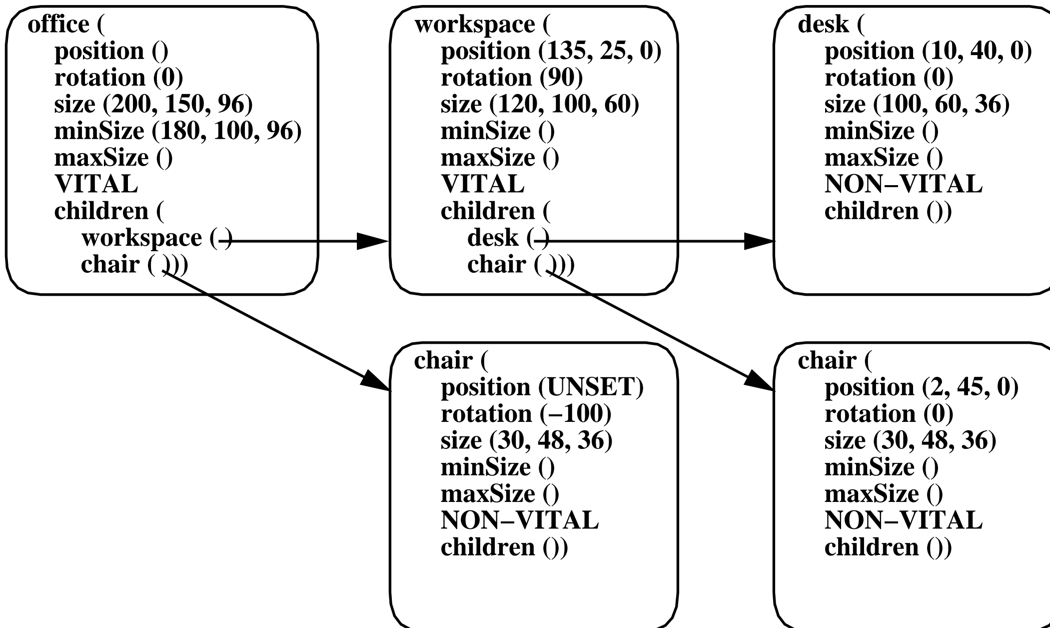


Figure 3-1: Diagram of office template.

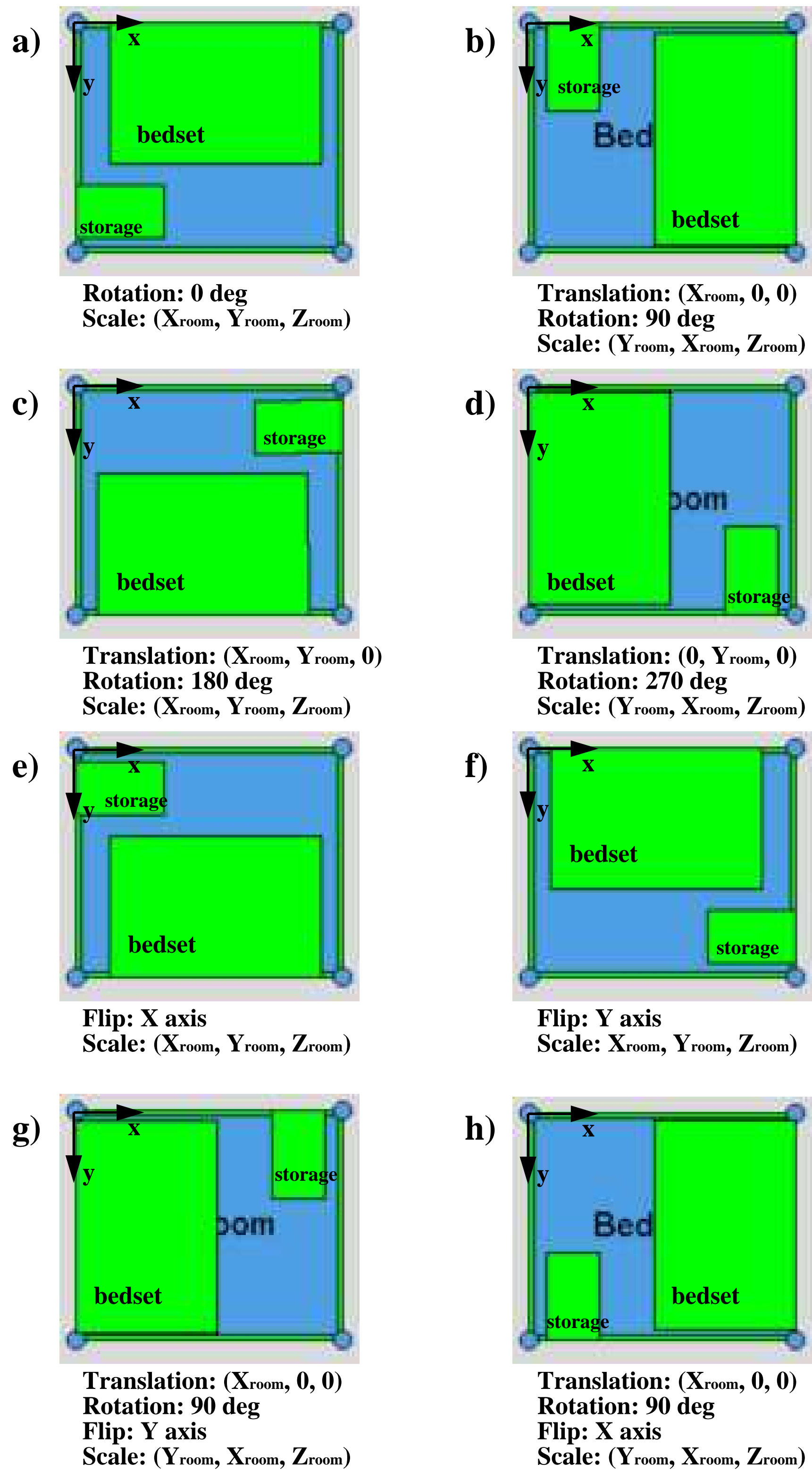


Figure 3-2: Possible mutations of room template.

3.2.1 Fitting templates to rooms

As previously discussed, all templates and rooms represented in FurnIt are shaped as rectangular solids. This makes it easy to fit a template to a given room. The three dimensions of the template are simply scaled to the width, depth and height of the room, respectively. However, there are still eight different ways the template can be oriented in relation to the room while maintaining the same floor plane (see Figure 3-2 on the page before). FurnIt initializes eight root templates for a given room, one for each possible orientation. These templates are mutations of the original template, obtained using the `Rotation`, `Translation`, `Flip` and `Scale` operators.

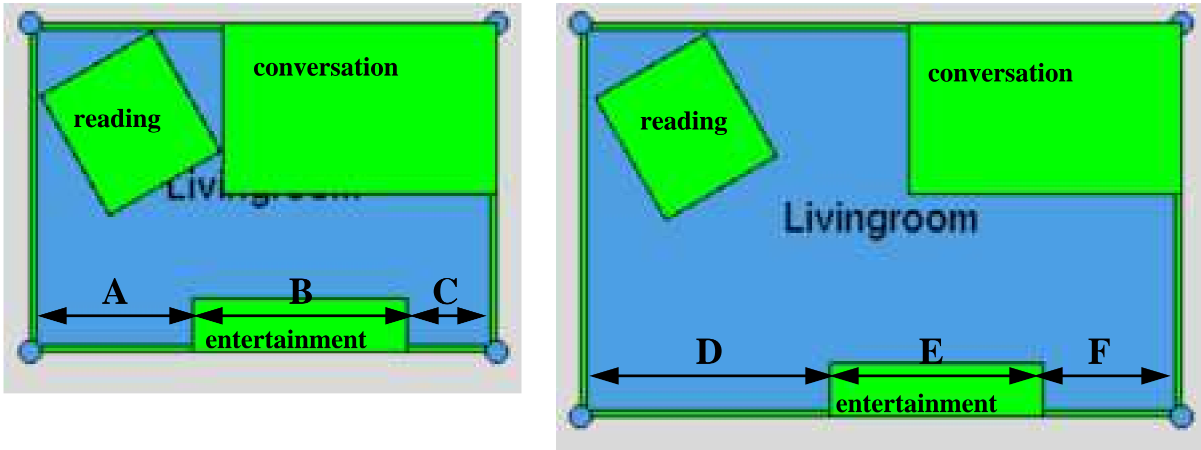
`Scale(w, d, h)` scales the width, depth and height of the template to give it the dimensions `(w, d, h)`. One root template can be created by simply scaling the width, depth and height of the template to the corresponding dimensions of the room (Figure 3-1 a, on page 19).

When a template is scaled, its children maintain their original sizes. (see Figure 3-3 on the following page). The position of the child templates are modified to conserve the configuration of the children, but their sizes remain constant. This is an intuitive result. The size of a piece of furniture is not scaled to be proportional to the room in which it is placed. One places additional pieces of furniture in a larger room, not the same number of larger pieces.

It is possible for a template to be scaled down so that it is smaller than one of its children. This can happen if the minimum size of the parent template is not set, or if its minimum size is set to be smaller than the given child. In this case, two outcomes are possible. If the child template is vital, the parent template is dropped from the computation. If the child is flagged as non-vital, it is dropped and the parent template is maintained without the child.

Even if none of the child templates are too large, scaling a template can lead to children interpenetrating. Techniques used for resolving these conflicts are discussed in Section 3.3.

`Rotation(angle)` simply rotates the template `angle` degrees around its origin. `Translation(x, y, z)` moves the template origin to the `(x, y, z)` coordinate in the parent coordinate system. Three new root templates can be created using these operators. The original template can be rotated by 90, 180 or 270 degrees, its origin translated and the template scaled appropriately in order to create three new root templates (Figure 3-1



$$\frac{A}{C} = \frac{D}{F}\,, \quad B = E$$

Figure 3-3: Scaling of room template.

b, c and d).

`Flip(axis)` reflects the child groups around a line at the center of the room along a given axis. Four additional root templates can be created with the `Flip` operator. The original template can be scaled, and its children flipped around the x- or y-axis (Figure 3-1, e and f, on page 19). Furthermore, the original template can be rotated 90 degrees and scaled before its children are flipped around either axis (Figure 3-1 g and h).

3.2.2 Doors and Windows

One requirement for a functional furniture configuration is that no doors or windows are blocked by furniture. FurnIt achieves this by placing empty child templates of appropriate height directly in front of all doors and windows (see Figure 3-4 on the next page). These templates are non-movable and non-scalable. Their purpose is to mark the space in front of doors and windows as "clear space", inaccessible to other child templates.

Once all root templates are computed, child templates marking "clear space" are added to each root template. This is easy since the location and size of each door and window is

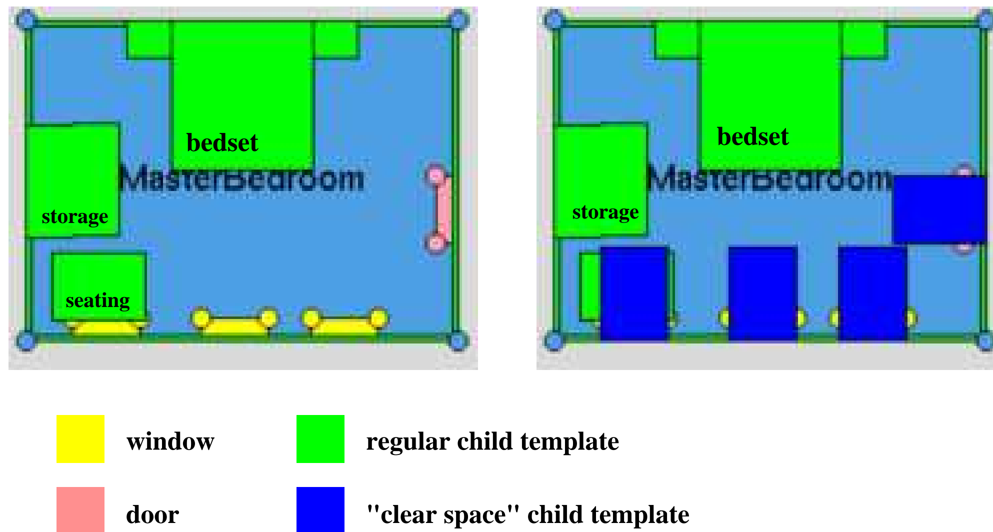


Figure 3-4: Adding "clear space" to a room template.

specified in the input floor plan. Note that the "clear space" template of a window occupies only the space directly in front of the window, not the space above or below it. This makes it possible to place furniture above or below a given window or above doors.

Adding "clear space" child templates can often lead to conflicts when these interpenetrate existing child templates. Interpenetration between child templates can also be the result of scaling a template to a smaller size. In many cases all root templates created for a given room will contain conflicting child templates. The next section discusses how these spatial conflicts are resolved.

3.3 Resolving Conflicts

The result of the initialization process is eight templates that are fitted to a given room, fewer if some fittings failed (see Section 3.2.1 on page 21). If any of the root templates contain no conflicting children, the system picks one of the resolved templates at random and tries to recursively resolve conflicts within its children (see Section 3.3.3 on page 27). Otherwise, the system picks a random template and tries to resolve conflicts between its children by moving, re-sizing and dropping child templates.

Note that while the child templates might interpenetrate, all child templates will be completely enclosed within the parent template. This is true of any valid original template, and all template operations used to create the root templates preserve this invariant. Thus we are guaranteed that no piece of furniture will penetrate the walls, ceiling or floor of the room. Only conflicts between child templates, possibly leading to interpenetrating pieces of furniture, need to be resolved.

The first step in resolving conflicts is to remove the child templates from the parent and sort them into three groups, "clear space", vital and non-vital. Any "clear space" children are added directly back into the parent template. These templates are non-transformable and no operations that help resolve conflicts can be applied to them. Also, interpenetration of "clear space" templates is not a problem since these are merely place holders marking open space. It is perfectly reasonable for a space to be unoccupied for more than one reason.

The next step is to add the vital child templates, one by one. For each child added, the system checks if any conflicts have arisen due to the new template. If so, the system tries to resolve these by moving or re-sizing the child templates. If a resolution can not be found, the root template fails and is removed from the computation.

The vital child templates are added to the parent in the order of their base area, the child occupying the largest area first. It is harder to place a large template than a small one. Placing the larger group first, while the room has more open space, makes the task easier and maximizes the chance of success. This approach also minimizes the computation time used on templates that will eventually fail.

Finally, the non-vital child templates are added. Like the vital child templates, these are placed in order of base area, and the same conflict resolution heuristics are used. However, failure to place a non-vital child does not lead to the failure of the root template. The system simply places as many non-vital children as it can; then it moves on to recursively resolve each child template (see Section 3.3.3 on page 27).

3.3.1 Moving Child Templates

FurnIt tries to resolve conflicts due to a newly placed child template using a standard set of heuristics. For each conflict, the system computes the axis-aligned bounding volume of the template intersection. It then tries to resolve the conflict by moving the newly placed child template out of this conflict volume without creating any new conflicts. The system will first

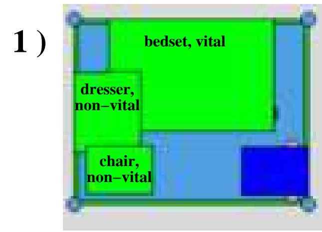
try to move the template along the minimum dimension of the conflict volume. If unable to resolve the conflict by this move, the system tries to move the conflicting template in this same dimension. If necessary, it will then try to move the new and conflicting templates along the conflict volume's other dimensions in order of magnitude. Note that since the conflict volume is axis-aligned, all moves will be axis-aligned also.

The directions in which a template can move are restricted by its location in the room. A template placed on the floor can not move vertically. Similarly, a child placed against a wall can move only along this wall; a child placed in a corner of its parent is restricted to be placed in corners. As mentioned before, all child templates must be contained within the parent template and their movements are restricted to the parent space. If the system tries to move a given child in a direction that conflicts with its location, the move will fail and the child remain in place. Finally, a child involved in multiple conflicts will not make any move to resolve one conflict that will increase the volume of any of its other conflicts.

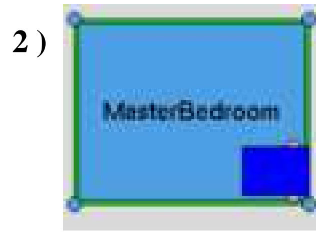
Even if the system is unable to resolve a conflict by moving a template, it will move the template in order to minimize the volume of the given conflict. If the given template is placed against a wall, the system will slide the template along this wall until it is either outside the conflict volume or unable to move farther. If the current template is placed in a corner, the system will try to place the template in the three other corners of the room. The template will be placed in the first corner where it causes no conflicts. If no such corner is found, the system will place the template in the corner with the smallest conflict volume. A child in the interior of a room is restricted to the floor plane, any surrounding child templates and by the room walls. It will move as far out of the conflict volume as possible by first attempting to move along the volume's minimum dimension, then attempting to move along each of the others if necessary.

Note that the system never tries to resolve conflicts by rotating child templates. While we want a set of operators sufficiently flexible to resolve conflicts, we also want to conserve the initial configuration of the child templates. If too wide a range of mutations is allowed, we have no guarantee that the final furniture configuration will be functional. FurnIt resolves this dilemma by allowing a limited range of translations only.

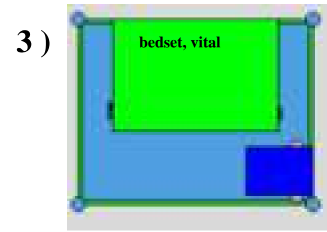
Figure 3-5 on the next page gives an example of a root template being resolved by moving its child templates.



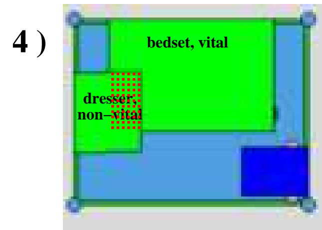
original root template



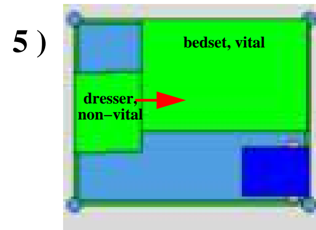
children removed and sorted,
"clear space" added back into tempate



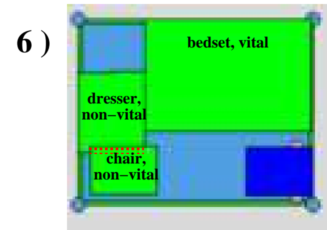
largest vital child added



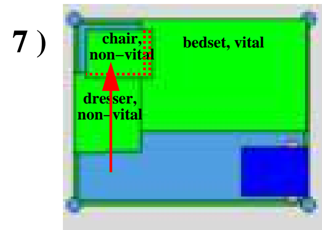
largest non−vital child added,
conflict volume computed



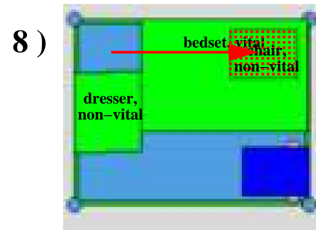
move in conflict volume's minimum
direction resolves conflict



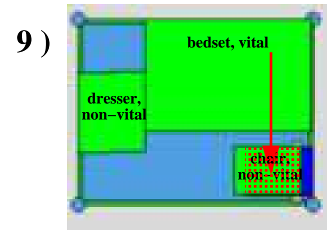
next non−vital child added in
corner, conflict volume computed



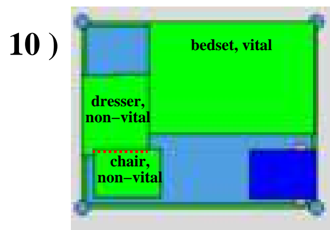
try 2nd corner,
conflict volume computed



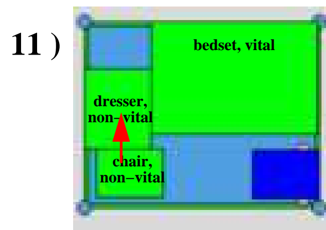
try 3rd corner,
conflict volume computed



try 4th corner,
conflict volume computed

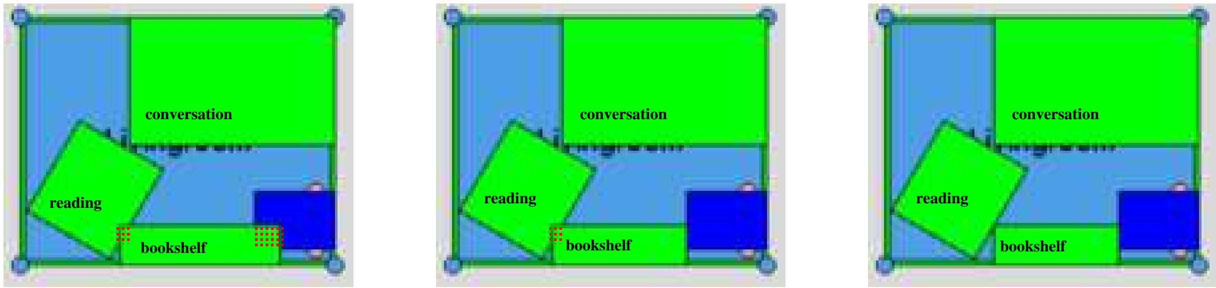


place in corner with
smallest conflict volume



move in conflict volume's minimum
direction resolves conflict

Figure 3-5: Resolving root template by moving children.



Bookshelf limited to move along wall. Unable to move due to multiple conflicts. Conflicts resolved by re–sizing template.

Figure 3-6: Re-sizing child template.

3.3.2 Re-sizing Child Templates

If the system is unsuccessful in resolving a given conflict by moving child templates, it will try to re-size the newly placed template. It first computes the maximum space available to the child in its current location. This is computed in the form of an axis aligned rectangular bounding volume similar to that of the conflict volume. The system will then fit the template into this volume by scaling and translating it appropriately. Note that the orientation of the template remains the same, and the restrictions on translations are the same as for a move (remain on floor, along wall, in corner). If the space is so small that it requires the child to be scaled below its minimum size, the child template is dropped. Dropping a template has two possible outcomes. If the dropped child was vital, the root template fails. If it was non-vital, the system simply moves on to the next child template.

Figure 3-6 gives an example of resizing a template that is unable to move due to multiple conflicts.

3.3.3 Recursion

Once the root template is resolved, the system recursively resolves each of its child templates. We have chosen a depth-first approach, where all descendants of the current child is resolved before the system tries to resolve the next child. Base cases are empty templates. These either represents "clear space" or single pieces of furniture.

If at any point a descendant of a root template fails to resolve, the root template fails. A different root template is chosen at random, and the system tries to resolve this. If the system is unable to resolve any root templates, the room remains empty.

The method for resolving children is the same as for resolving root templates, except that child templates do not contain "clear space" due to doors and windows. It is, however, possible for a user to define a child template that contains "clear space" if this is necessary for defining a desired furniture configuration.

Unlike the room template, the orientation of most child templates are fixed. If a "reading" template containing a chair and a side-table is placed in the corner of a room, we want this child to retain its original orientation and not end up facing the corner. Thus the eight mutations described earlier are inappropriate for most child templates. We accommodate this by allowing templates to be flagged as `fixed`. If this `fixed` bit is set, the system will not attempt to compute all eight orientations of the template, but simply scale the template to fit its parent.

There is one other subtle difference between a root and a child template. In a root template, all corners are in the meeting point of two walls. This is not necessarily true for child templates. Thus the heuristic of trying to place a corner child in all corners is not appropriate. Instead, the system allows corner children of child templates to move along either edge forming the corner.

3.3.4 Base Cases

FurnIt recursively resolves geometrical inconsistencies for a given room. The base case is an empty template. These "leaf templates" either mark clear space or a single piece of furniture. Once a given root template is completely resolved, the system places furniture in those leaf templates that do not represent clear space. The furniture type placed in each template is determined by the template type (e.g. a chair is placed in a "seating" template). Procedures for placing furniture in all default template types are provided. If the user defines a new template type, she must also provide a procedure for placing furniture in the this template type.

Some default template types can be satisfied by more than one furniture type; a "storage" template, for example, can be filled by either a dresser or an armoire. In this case the template's furniture placement procedure will determine which type of furniture to place in

the template. In most cases this decision is based on the template size.

Note that most of the leaf templates are not the size of the piece of furniture they represent; rather they are the size of the space needed to use the given piece of furniture. A "seating" template contains sufficient space for the chair itself as well as for the user's legs in front of the chair.

Table A.3 on page 60 lists all default leaf template types. It specifies which templates include additional space. It also gives common examples of what furniture types the system places in each default template.

Chapter 4

Implementation

FurnIt was implemented in C++ [16] as part of the Fire Walk [18] system. The template data structures were implemented as a set of C++ classes within SYLIF (SYmbolic Layout Interchange Format), a framework for representing building information developed by Laura Downs [8]. The FurnIt user interface is an extension of the existing Floorsketch interface, which was written by Rick Bukowski using Tcl/Tk [15]. The FurnIt control procedures and conflict resolution procedures were implemented as a stand-alone library by the author.

This chapter will discuss the implementation of each component of the FurnIt system and describe in detail how FurnIt interacts with other components of the Fire Walk system.

4.1 FurnIt Control Flow

The FurnIt control and conflict resolution procedures are implemented as a C++ library. The entry point to FurnIt is a top-level control procedure. It takes as input a SYLIF data base and extracts a list of the rooms contained in its floor plan. It then iterates through this list, choosing a template for each room based on the room label. Rooms with unknown labels are ignored and left empty. Next, FurnIt fits the chosen template to the room as previously discussed. Note that the resulting list contains the resulting root templates in random order. This is done to avoid choosing the same template in all cases where the root templates contains no conflicts.

FurnIt next iterates through the list of root templates, checking for templates with no conflicts. If one is found, the program tries to recursively resolve its children using the conflict resolution heuristics previously discussed. If no conflict-free template is found, the

program iterates through the list of room templates again, this time trying to resolve any conflicts in each root template. If successful, the program attempts to recursively resolve the given template's children.

If the program is unable to resolve any of the root templates, the room is left empty and FurnIt moves on to the next room in the list. If a solution is found, furniture is placed in the corresponding room. One piece of furniture is placed for each leaf template that does not represent clear space. Each leaf template type has a procedure that places a piece of furniture corresponding to the template. This procedure determines the type, size, location and rotation of the furniture object based on the leaf template.

4.2 FurnIt Debugging Interface

FurnIt has an extensive debugging interface. It is difficult to test new templates and conflict resolution procedures without being able to view each step of the conflict resolution process. A graphical interface is desirable, since it is difficult to visualize how spaces relate to each other based merely on textual output.

In debugging mode, FurnIt displays the current template configuration for each step in the conflict resolution process. Figures 3-2, 3-3, 3-4 and 3-5 were created using this interface (some are augmented with arrows etc.). Icons representing each template are dropped into the Floorsketch window (see Section 4.4 on page 39). Each time FurnIt checks or modifies a template, the `modified` flag for this template is set to true. If the `modified` flag of a template is not set, this template is displayed as a unit (instead of recursing down and displaying its children). This is done so that the user can view the template hierarchy at the level at which the program is currently working.

When FurnIt has iterated through all rooms in the floor plan's room list, the augmented SYLIF data base is returned.

4.3 SYLIF (SYmbolic Layout Interchange Format)

SYLIF, an ASCII file format and a set of C++ data structures, was developed to encode both the specifications for a proposed building and symbolic layouts for floor plans that might satisfy these specifications [8]. It was originally developed to allow clients to formally

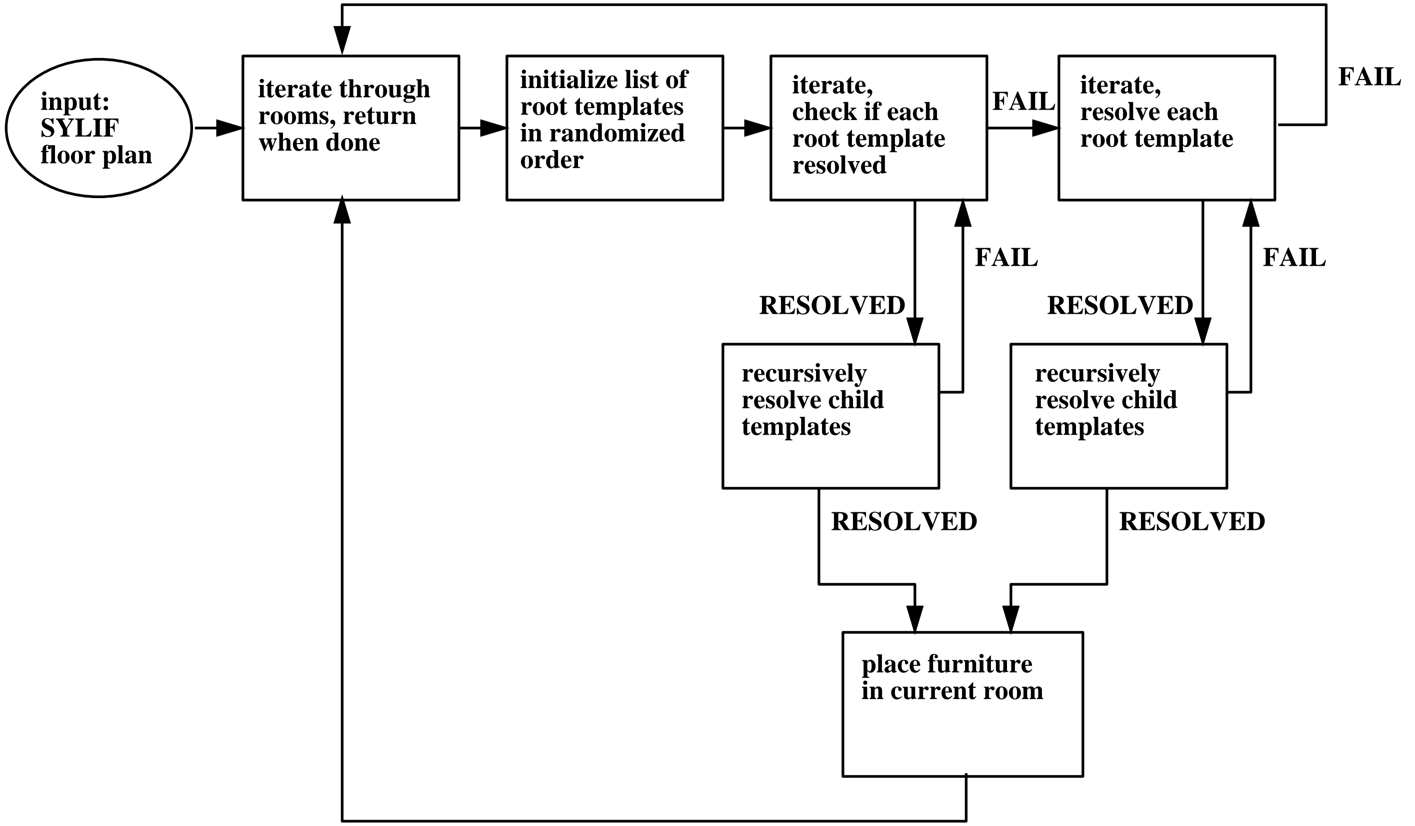


Figure 4-1: Control flow in FurnIt.

specify design requirements and to allow architects to evaluate potential designs with respect to these requirements.

FurnIt takes as input a floor plan in the form of a SYLIF data base. It outputs the same data base augmented with SYLIF furniture objects. Furthermore, the template data structures used by FurnIt are implemented as SYLIF classes. This section presents a brief overview of the SYLIF framework and discusses the implementation of the template data structures.

4.3.1 Overview of SYLIF Data Structures

The SYLIF data structures are implemented as a C++ library, using STL (Standard Template Library). The library includes a parser that is generated using the `bison` and `flex` [11] parser generations tools. The parser reads a SYLIF file, creating a top-level data base that contains all information from the given file. Furthermore, all data structures are equipped with an output function. The output function for a given data structure writes all information contained in that structure into a file.

The SYLIF file language is designed to be comprehensible to humans. Its syntax is too complex to be discussed here, and the user is referred to Downs [8] for a complete description.

A SYLIF data base consists of three main data structures: `BuildingInfo`, `BuildingProgram` and `BuildingLayout`. The `BuildingInfo` data structure defines units and standards used throughout the building. It also defines generic building components, such as default doors and windows, that can be referenced later in the building description. Note that these default components can be replaced by custom data structures at later points in the program. Finally, the `BuildingInfo` contains information pertinent to the building as a whole, such as a project or building name, the location of the building site, the elevation of the building above sea level, etc.

The `BuildingProgram` outlines the desired contents of the building. It lists the room types that should appear in the building, specifies the total area that should be dedicated to each room type, indicates how many rooms there should be of each type and specifies how rooms of different types should be placed in relation to each other. Several data structures are used to express these relationships, and the reader is referred to Downs [8] for a comprehensive discussion. Here it is sufficient to note that several mechanisms are

available to specify the desired building contents, and that the user is free to use any subset of these.

The final component of a SYLIF data base is the `BuildingLayout`. This contains a symbolic representation of the building floor plan. Each room has a size, location and label; window and door locations are also specified. Note that the room geometry in SYLIF is only 2 1/2 dimensional. This means that while the x- and y-dimensions are completely described, information regarding the z-dimension of any object is given only as an elevation.

4.3.2 SYLIF-FurnIt Interaction

SYLIF data structures serve as a common format used to pass building information between City Walk applications. A number of these applications has been enhanced so that they can read and write SYLIF files. This enhancement is made easy due to the SYLIF parser and output functions.

The input to FurnIt is a SYLIF data base describing a given building. FurnIt is only concerned with the `BuildingLayout` component of the data base. It does not try to evaluate the overall building design, but interacts with the data base on the room level, iterating through each room in the data base and populating it with furniture. This is a reasonable heuristic, given that the furnishing of a room is not influenced by the overall building layout.

FurnIt outputs a complete SYLIF database containing SYLIF furniture objects. These are defined by a size (of the bounding box), translation, rotation and type. None of the structural components of the data base are modified by FurnIt. The only difference between the input and output data base is the added furniture objects.

4.3.3 SYLIF Template Classes

The FurnIt templates were implemented as SYLIF data structures for three reasons. First, we wanted to take advantage of the existing SYLIF infrastructure and the SYLIF parser. Secondly, we wanted to take the opportunity to use SYLIF as a common format for indoor data, shared by several City Walk applications. Finally, we wanted to enrich the SYLIF data base format with a compact representation for how rooms should be furnished. The default furniture templates are created whenever a SYLIF data base is initialized. In addition to these default templates, the user can create custom templates that specify how a given room or room type should be furnished. Both default and custom templates can be passed

around between applications as part of the building data base until it is time to populate the building with furniture.

Figure 4-2 on the next page shows the complete class inheritance hierarchy of SYLIF. The following is not a complete discussion of this hierarchy; rather it is a description of only the classes used to implement templates and how these classes relate to the overall SYLIF framework. Note that while SYLIF was implemented by Laura Downs, all classes implementing templates were written by this author. It was also necessary for this author to significantly extend the existing furniture classes, as well as to extend the overall SYLIF framework to include the new data types.

All SYLIF objects are derived from the SLFBase class. This class supports basic functionality, such as creating, initializing and copying objects, and reading and writing SYLIF files.

SLFTemplateType

All objects with a string identifier and label are descended from the SLFEntity class. The SLFTemplateType class is derived from this class. The string identifiers allow template types to be referenced from other objects in a SYLIF file. This mechanism can be used to specify which template should be used to populate a given room or room type.

Template types are organized in a SLFTemplateTable that supports location by id and insertion and deletion of template types.

SLFTemplateInfo

Each SLFEntity has an associated SLFInfo object. This info object holds a wide range of information specific to the entity. The SLFInfo class is separated from the SLFEntity class to allow for extentions to the information associated with an object type. In order to associate additional information with an object type, a user can define a subclass of the SLFInfo class. Thus a data type can be extended without modifying the inheritance hierarchy of the SYLIF libraries.

The SLFTemplateInfo class is derived from the SLFInfo class, and each template type has a reference to a template info object. This info object contains all information common

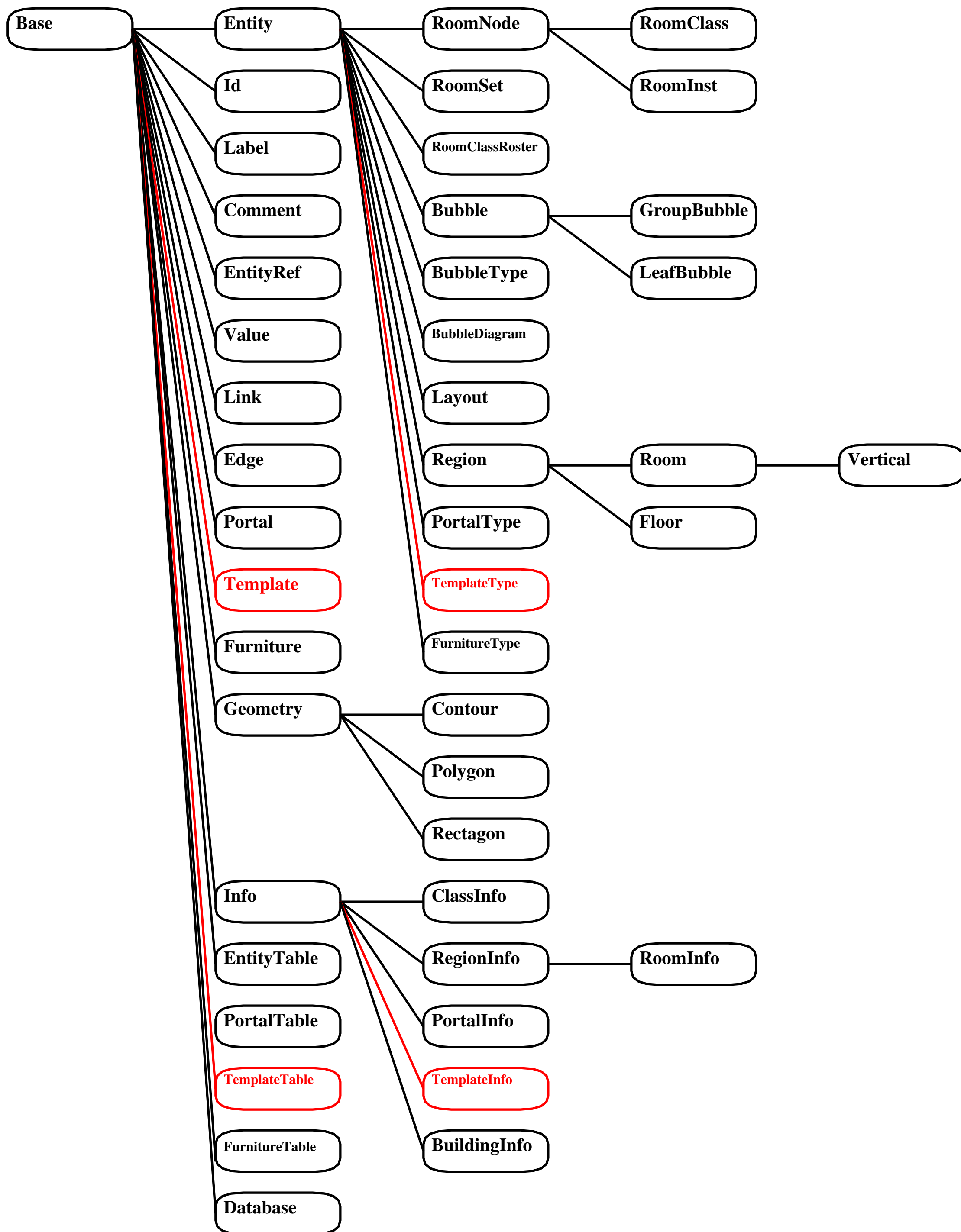
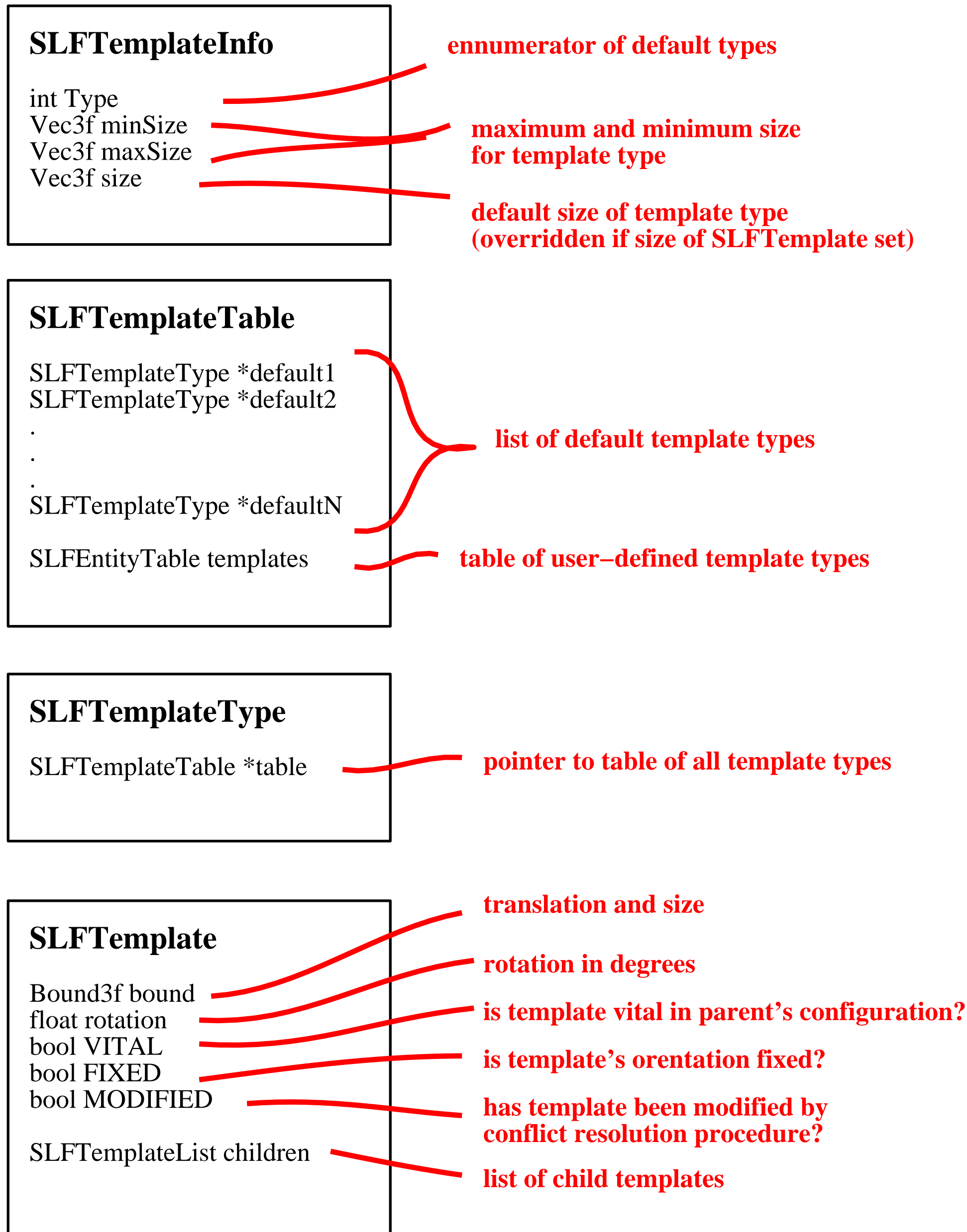


Figure 4-2: SYLIF class inheritance hierarchy. Adapted from Downs [8].



Note that a C++ class inherits the member variables of its parent class,
e.g. SLFTemplateType inherits a reference to the info object from SLFEntity.

Figure 4-3: Member variables of SYLIF template classes.

to templates of a given type (see Figure 4-3 on the preceding page).

SLFTemplateTable

The SLFTemplateTable class is derived from the SLFEntityTable class. It organizes all template types in a global hash table. This allows fast location by id, insertion and deletion of template types. It also avoids redundancy by making it impossible to define more than one template type with the same identifier. This is an important feature since users are allowed to define new template types.

SLFTemplate

SLFTemplate is a single instance of a template type. It contains a reference to a type, as well as a size, location, rotation and a list of children. It also contains the booleans `vital`, `fixed` and `modified`. `Modified` is set by the conflict resolution procedures when they *begin* to resolve the template. It is used for debugging and display purposes (see Section 4.1 on page 30).

The SLFTemplate class provides access functions for its member variables. Note no access function is given for template size. A given template must be scaled to a given size using the `Scale` function in order to ensure that all children of the template remain enclosed in their parent.

Furthermore, the SLFTemplate class provides functions for adding, removing and accessing the children of a given template. However, no function for accessing a template's parent is provided. The FurnIt algorithm resolves a given space top-down, resolving each template *before* attempting to resolve its children. Thus it is not necessary to backtrack during the computation, and no parent access function is necessary.

In addition to the above mentioned access functions, the SLFTemplate class provides a wide range of functions used for reasoning about spaces. Below is a list of some of the procedures provided.

- `Bool Contains(SLFTemplate *other)` returns true if current template contains other, false otherwise.

- `Bool Intersects(SLFTemplate *other)` returns true if current template intersects other, false otherwise.

- `Bool Resolved()` returns true if no children intersect each other, false otherwise.
- `Bound3f GetIntersection(SLFTemplate *other)` returns the axis aligned bounding volume of the template intersection.
- `Bound3f GetUnion(SLFTemplate *other)` returns the axis aligned bounding volume of the union of the templates.
- `Bound3f GetBoundingBox()` returns the axis aligned bounding volume of the template.
- `Bound3f GetMinBoundingBox()` returns the axis aligned bounding volume of the template if scaled to minimum size.
- `Bound3f GetMaxBoundingBox()` returns the axis aligned bounding volume of the template if scaled to maximum size.
- `Polygon2f* GetPolygon()` returns a 2D footprint of the template base.

4.4 Floorsketch

Floorsketch is a tool for quickly drawing a floor layout with doors and windows (see Figure 4-4 on the following page). Floorsketch displays the floor layout in 2D, and allows the user to interact with the x- and y-dimensions of the layout elements using a click-and-drag interface. However, the internal data representation of Floorsketch is 2 1/2 dimensional, and the user can define parameters such as ceiling height, door height, sill height of windows and floor offset from ground level.

Floorsketch is implemented as a set of Tcl/Tk data structures and a set of C++ callbacks that make it possible for other applications to interact with these data structures. Translators between the Floorsketch data structures and SYLIF exist, and Floorsketch is capable of reading and writing SYLIF files. As mentioned previously, Floorsketch interacts with FurnIt by passing it a SYLIF data base representing the current floor layout. FurnIt returns this data base augmented with furniture objects, but is also able to place furniture icons directly into Floorsketch using the provided C++ callbacks.

Floorsketch is equipped with error-checking methods that ensure that the floor layout drawn by the user is geometrically valid. This ensures that all rooms are adjacent, that

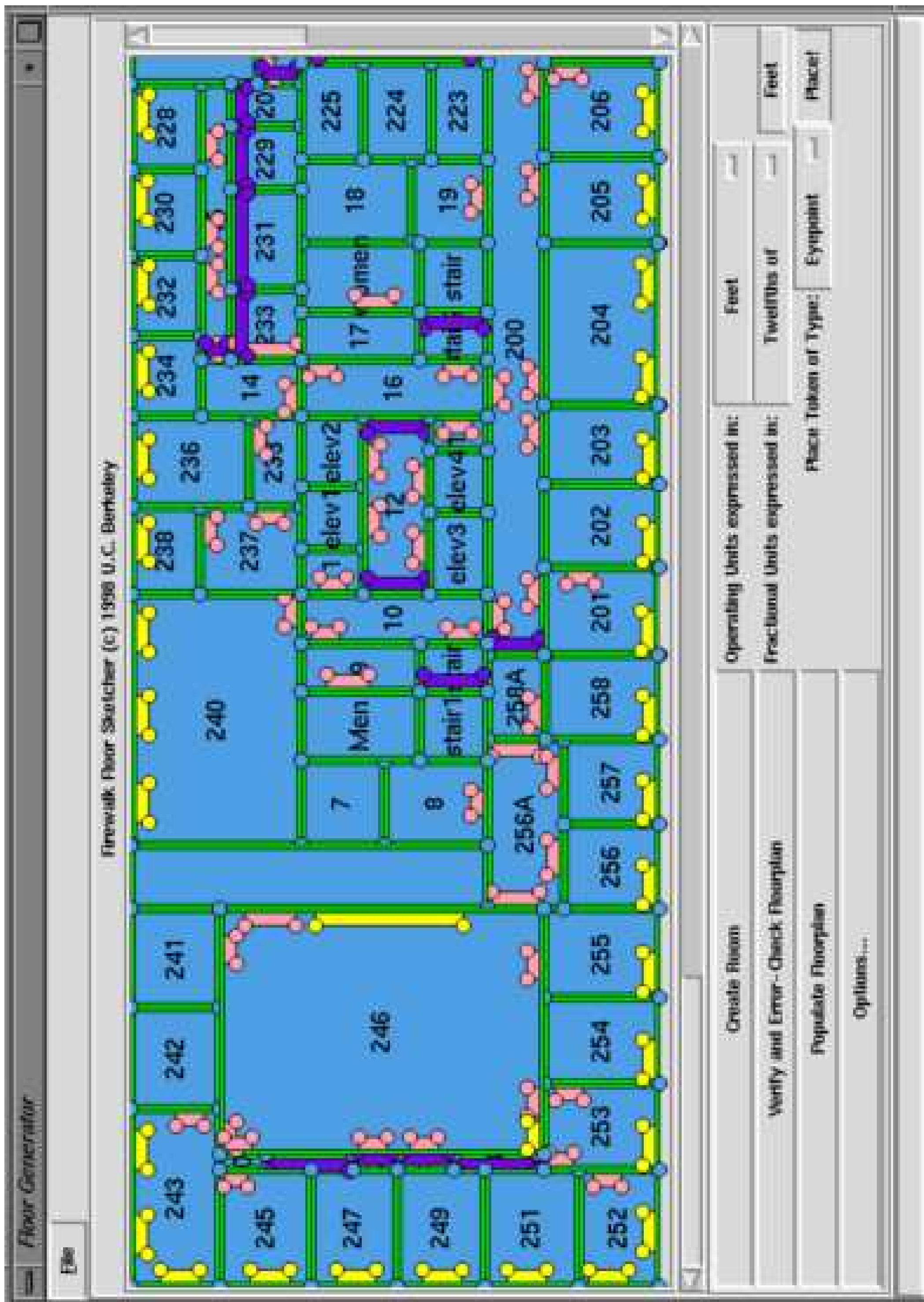


Figure 4-4: Floorsketch view of second floor of building NE43.

doors and windows are in valid positions, etc. The Floorsketch error-checker is used by FurnIt. SYLIF does not guarantee that the floor layout represented by its `BuildingLayout` object is geometrically valid. Inaccurate layouts are deliberately allowed, making SYLIF more suitable for use on early and intermediate designs for a building. FurnIt, on the other hand, assumes that its input is geometrically valid. Thus the Floorsketch error checking procedures are necessary to ensure geometrical consistency of a given floor plan before attempting to populate it with furniture. This ensures that only geometrically valid SYLIF data bases are exported from Floorsketch and passed to FurnIt.

Floorsketch is capable of exporting its contents to the wk file format used by the WALKTHRU viewer. The Floorsketch layout is translated to SYLIF, then from SYLIF to Unigraphics, an ASCII file format used to represent 3D geometry. Note that this latter conversion involves extrusion from 2 1/2 dimensions to three dimensions. Floorsketch provides sufficient information to make this possible. The data is then translated from Unigraphics to the wk file format, suitable to be viewed with the WALKTHRU viewer.

4.4.1 Extensions

Several extensions were made to the Floorsketch interface in order to extend it to provide the FurnIt user interface as well. First, a button for furniture population was added to the Floorsketch window. This button simply translates the floor layout currently displayed in the Floorsketch window to SYLIF and calls the FurnIt program on this data base.

Furniture placed by FurnIt is displayed as rectangular icons with the appropriate size and location in the Floorsketch floor layout. The user can move and rotate these icons by hand if she wishes to modify the configuration generated by FurnIt. A pull-down menu for placing additional pieces of furniture into the floor layout by hand has also been added.

FurnIt requires a room to have a type or label that informs FurnIt of the room's function. In order to provide this, we added a pull-down menu to Floorsketch that allows the user to specify a label for each room. Rooms have the default label "unlabeled" when created. The Floorsketch labels are propagated into SYLIF and read by FurnIt. Rooms labeled with the default label or unknown labels are ignored and left unfurnished.

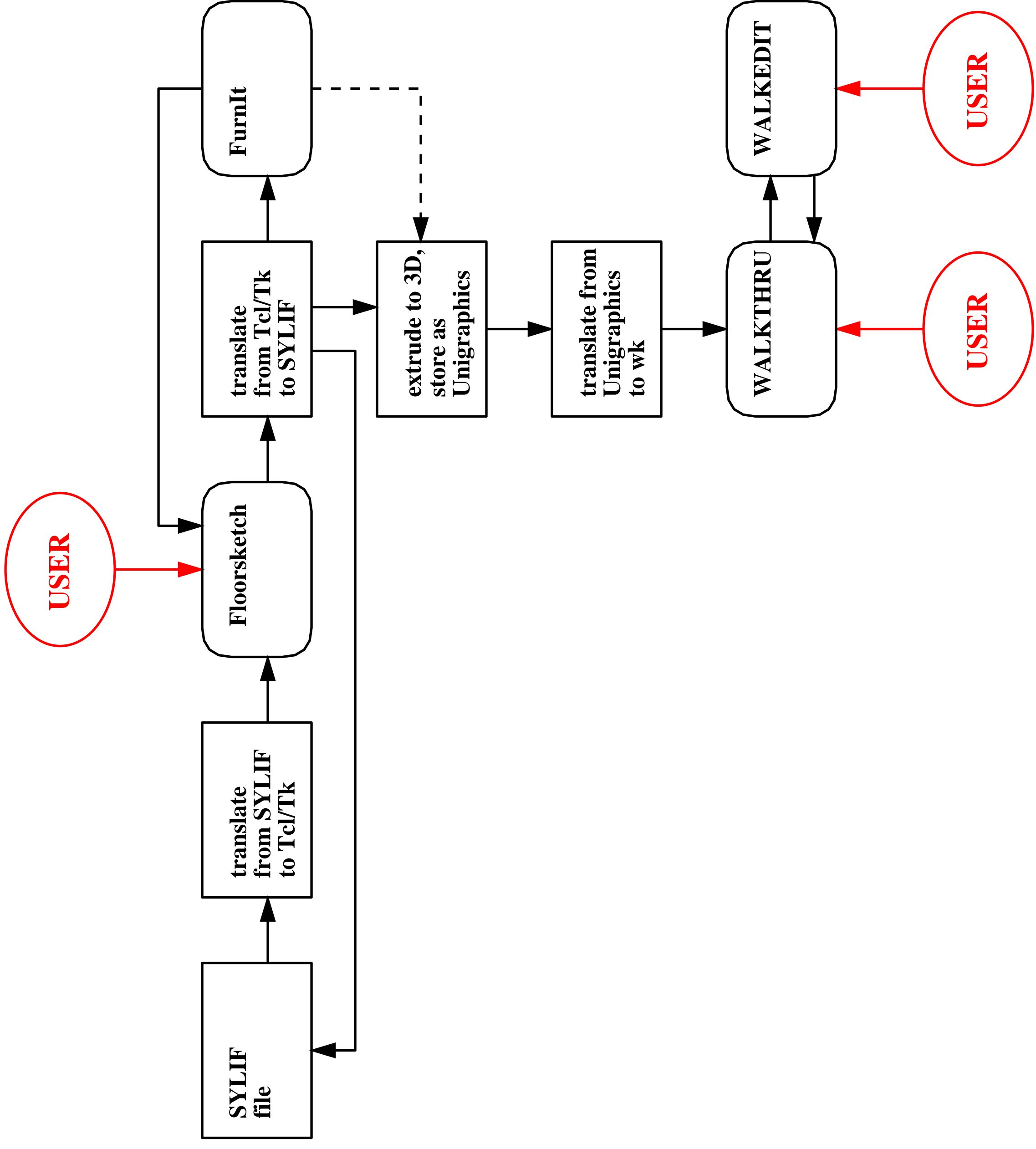


Figure 4-5: Interactions between City Walk applications.

Chapter 5

Results

This chapter presents instructions for how to use FurnIt, as well as results achieved with the FurnIt system. All default room templates are depicted and described. The variety of results that can be achieved using FurnIt is demonstrated by populating a simple floor plan five times using the same templates, each time resulting in a different furniture configuration.

5.1 Using FurnIt

The following is a set of detailed instructions for how to use Floorsketch and FurnIt. Together these applications enable the user to create a floor plan (with room labels and door and window information) that is guaranteed to be geometrically valid and populated with furniture in a reasonable configuration.

5.1.1 Creating Rooms

The user creates a room in Floorsketch by pressing the "Create Room" button in the Floorsketch main window. The new room has a default size and a default label "unlabeled". The room can be resized by clicking and dragging one of its corners. The room can be moved by clicking inside the room and dragging the room to its desired location. Note that room edges and corners snap to each other, making it easy to tightly pack rooms in a floor plan.

The user can access all room parameters by double-clicking inside a room, bringing up the room menu window. This window contains a button for destroying the room, as well as fields allowing the user to type in exact values for the room position and size. Furthermore, the room window contains a pull-down menu that allows the user to specify the label of the

room from a list of available labels. Note that after the user selects a label, she needs to press the “change label” button in order for the change to take effect.

5.1.2 Adding Portals

In Floorsketch, a wall opening is called a portal. Available wall opening or portal types are doors, windows and wall-sized openings where an entire wall is removed. Each portal type has a default size. Windows also have a default height above the floor level. Each portal is color coded according to its type. Doors are pink, windows are yellow and wall-sized openings are purple.

Portals are created by double-clicking along the wall of a room, specifying the new portal’s location. The default portal type is a door. Portals can be moved by clicking on the portal and dragging it along the wall. It is not possible to move a portal off the wall on which it was created. A portal can be resized by clicking on its corner and dragging it until the portal has the desired width.

Double-clicking on a portal brings up the portal menu window. This can be used to destroy a portal, or to change its type to a door, a window or a wall-sized opening. The elevation and height of the portal opening can also be set.

5.1.3 Furnishing a Floor Plan

Once the user has created a floor plan using labeled rooms and portals, this floor plan can be automatically furnished. Note that before the user attempts to furnish a given floor plan, she should verify that it is geometrically valid by pressing the “Verify and Error-Check Floor Plan” button in the main Floorsketch window. This will ensure that no rooms overlap, that all portals are valid (not larger than the wall in which it is placed) and that no portal faces a T-joint, i.e. opens onto a perpendicular wall. Floorsketch will report any errors found, and the user can then remedy these before attempting to populate the floor plan with furniture.

To furnish a floor plan, the user simply presses the “Populate Floor Plan” button in the main Floorsketch window. The user will then be prompted by FurnIt, asking if she wants to view each step of the furnishing process. Choosing this option will cause FurnIt to display the current template configuration after each step in the mutation and conflict resolution procedures. For each step, the program prompts the user, asking whether she

wants to continue to view each step, or if she wants to move on to view the finished furniture configuration.

FurnIt places icons representing pieces of furniture into the floor plan displayed in the Floorsketch main window. These icons have the size and position of the furniture they represent, and are color coded according to function. Seating (chairs, sofas etc.) is pink, tables are brown, shelves are beige, storage is (dressers, armoires) grey and beds, kitchen and bathroom furnishings are white. Icons representing clear space are blue and icons representing groups of furniture are green. Blue and green icons are only displayed in step-mode, to allow the user to follow the program's reasoning about high-level templates. See Table A.4 on page 61 for a complete list of icons and colors.

5.1.4 Manually Adjusting Furniture Configuration

Once the floor plan is furnished, the user can manually adjust the position and rotation of any piece of furniture, as well as add new pieces of furniture to the floor plan.

A piece of furniture can be moved by clicking on its icon with the left mouse button and dragging it to its new location. The icon can be rotated around its origin by clicking on it with the middle mouse button and moving the mouse. New furniture can be placed using a pull-down menu of default furniture types found in the main Floorsketch window. Once a furniture type is selected, the user clicks the "Place!" button and then clicks at the location in the floor plan where the icon should be placed. Note that furniture icons can only be placed inside rooms.

5.2 Default Templates

There are six default room templates provided by FurnIt: living room, dining room, master bedroom, bedroom, office and student office. In addition to these, several default templates for furniture groups and individual pieces of furniture are provided (see the second and third tables in Appendix A for a complete listing). These can be used as building blocks when creating custom templates. The range of templates provided by FurnIt reflects that the system was originally designed to furnish models of private homes. However, it has been extended with templates appropriate for faculty and student offices, making it suitable for a wider range of building types.

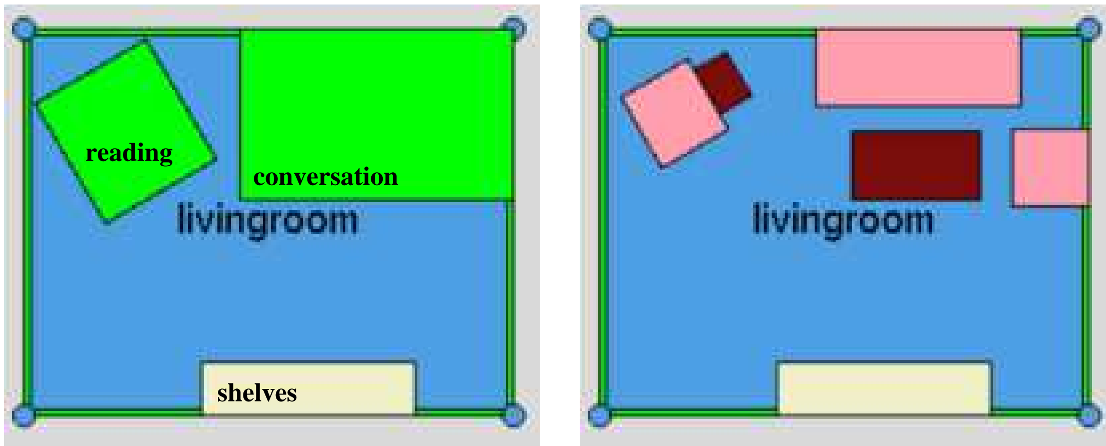


Figure 5-1: Default living room template.

Significant amount of time was put into ensuring that templates for individual pieces and groups of furniture contained sufficient open space to function in the parent template [13]. For example, a template for a dresser contains sufficient room for the drawers to open. A template for a dining room set contains sufficient space for the chairs to be pushed back from the table. Thus, piecewise functionality is ensured by the child templates, and all that is left is to combine these in a reasonable configuration.

The FurnIt default templates model their configurations on common furniture configurations found in homes and offices. The default templates represent the most commonly found room types. The user is provided with an interface for creating additional custom templates as needed.

5.2.1 Living Room

The default living room template (see Figure 5-1) contains a conversation group (sofa, table and chair), a reading group (table and chair) as well as shelves. Only the conversation group is vital. Note that the shelves are visible from all furniture groups in the living room, which also allows the shelves to serve as an entertainment unit, containing a TV set.

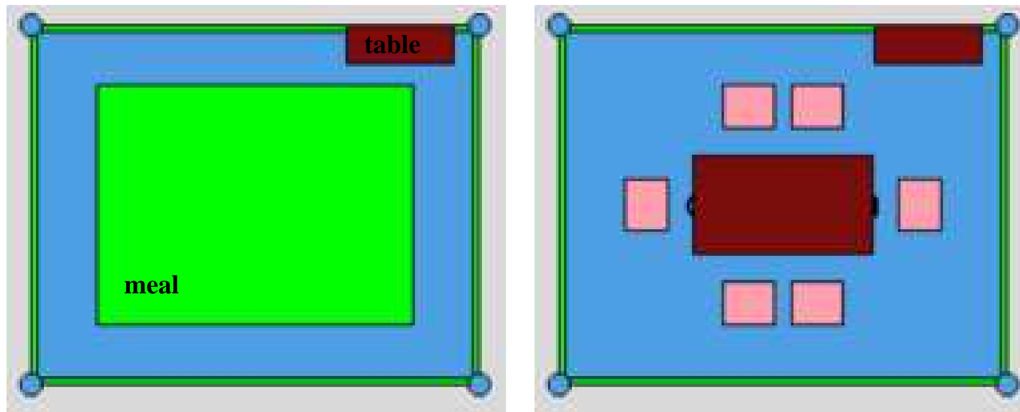


Figure 5-2: Default dining room template.

5.2.2 Dining Room

The default dining room template (see Figure 5-2) contains a dining room set and a side board. Only the dining room set is vital. Note that the dining room set template contains sufficient space for chairs to be pushed back from the table.

5.2.3 Master Bedroom and Bedroom

Default templates for both master bedrooms and bedrooms are provided (see Figure 5-3 and 5-4). The master bedroom contains an armoire and seating, the bedroom contains a dresser. Both templates contain bed sets consisting of a bed and its accompanying bedside tables. The bed set is the only vital child of each room template.

5.2.4 Office and Student Office

The default office template (see Figure 5-5) is modeled after MIT faculty offices. A workspace (desk and chair), extra table space and shelves are provided. The workspace template is vital and scaled to its maximum size.

The default student office template (see Figure 5-6) is modeled after the graduate student offices in the MIT Laboratory of Computer Science. The template contains workspace for three students. However, only two of these are vital, and the third desk and chair will be dropped if the room is too small to accommodate three students. The workspace templates

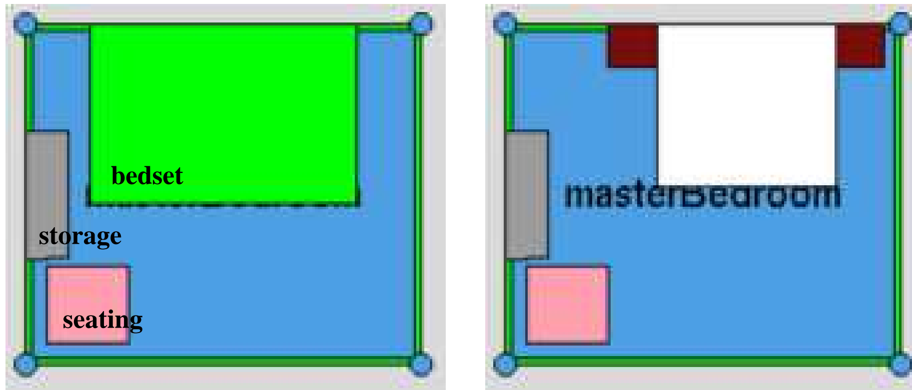


Figure 5-3: Default master bedroom template.

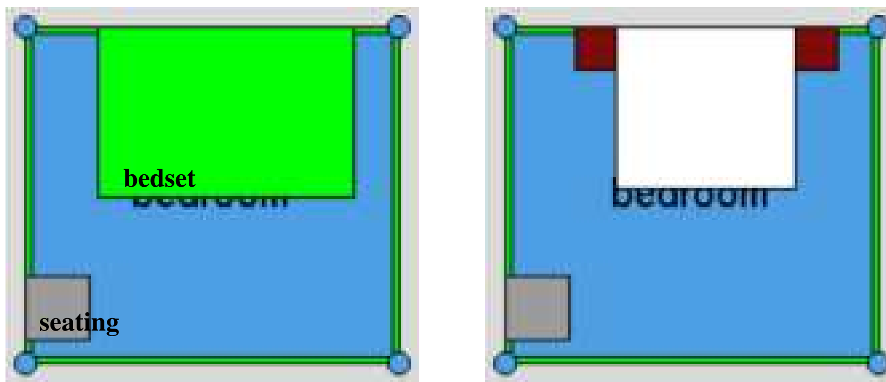


Figure 5-4: Default bedroom template.

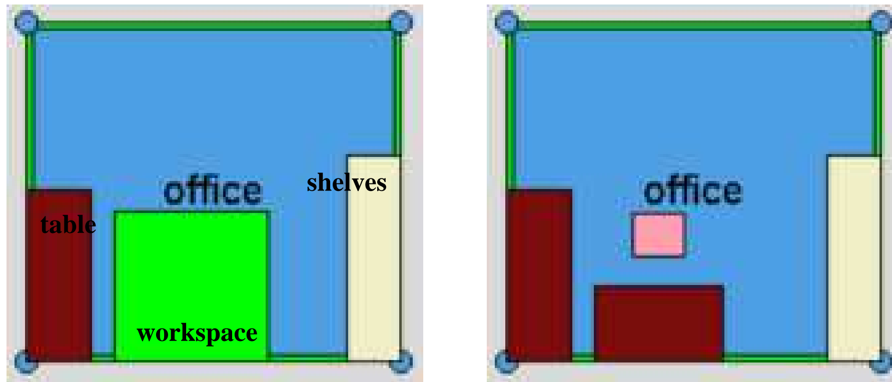


Figure 5-5: Default faculty office template.

are scaled to their minimum size.

Note that the shelf in the student office is hanging from the wall. It fits above the workspace of one of the students, and does not conflict with the workspace template, although it might appear so in Figure 5-6.

5.3 Variations in Furniture Configurations

As previously discussed, FurnIt computes the root templates for a given room in random order. This means that a random mutation of the original template is chosen in cases where no initial conflicts are detected. Furthermore, conflicts differ between mutations, since each mutation has different conflicts with clearspace templates due to doors and windows. This leads to different conflict resolution procedures being applied to different mutations of the same template. The result is that a range of furniture configurations can be created from a single template. For a floor plan with several rooms, the chance that all rooms will have the same furniture configuration for several consecutive runs of FurnIt is very small.

A simple floor plan incorporating all the default template types is depicted on the following pages. This example floor plan was populated five times, each time with a different result. Some rooms have the same furniture configuration more than once. For example, the dining room has the same configuration in all cases. This is due to the dining room being small compared to the applied template. Scaling the template down leads to all but one

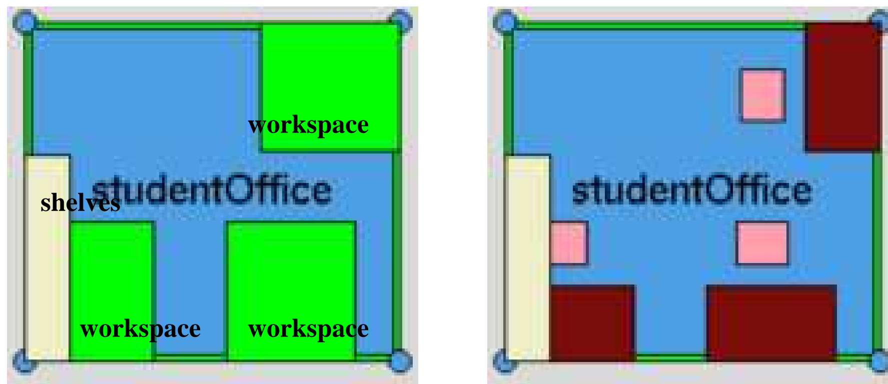


Figure 5-6: Default student office template.

root template failing, and the result is that the room has the same furniture configuration in all cases.

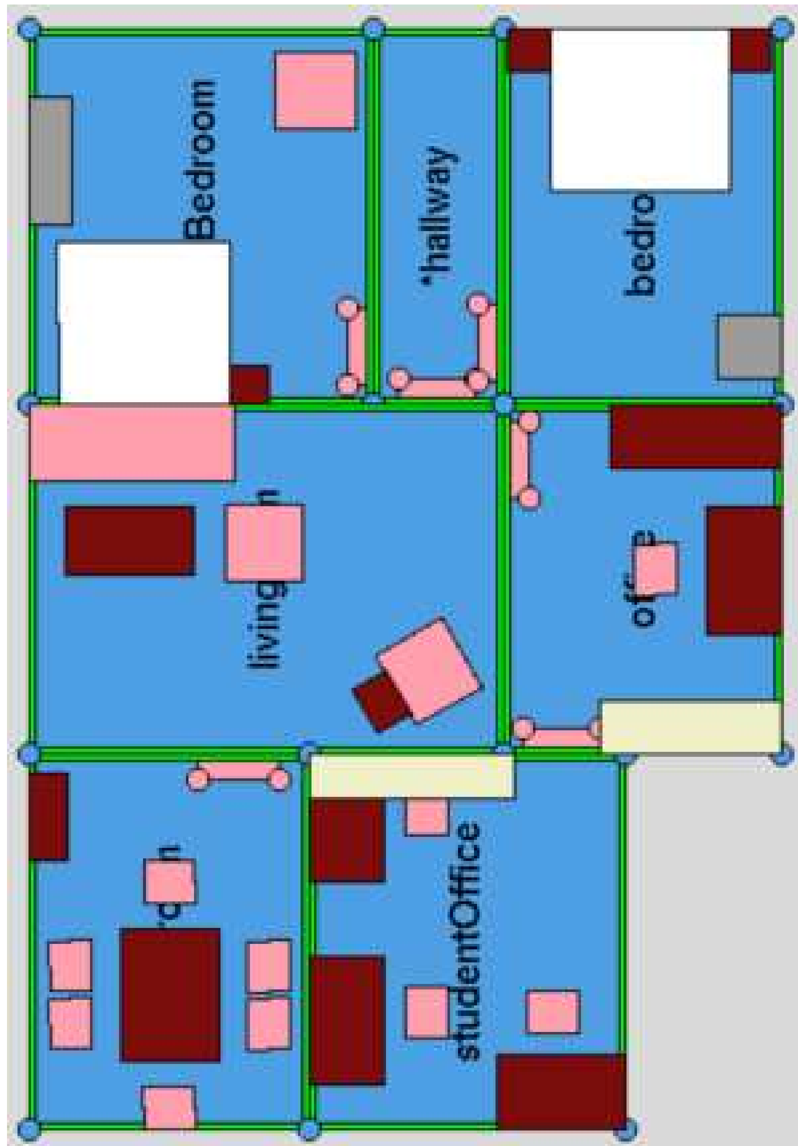


Figure 5-7: First sample configuration.

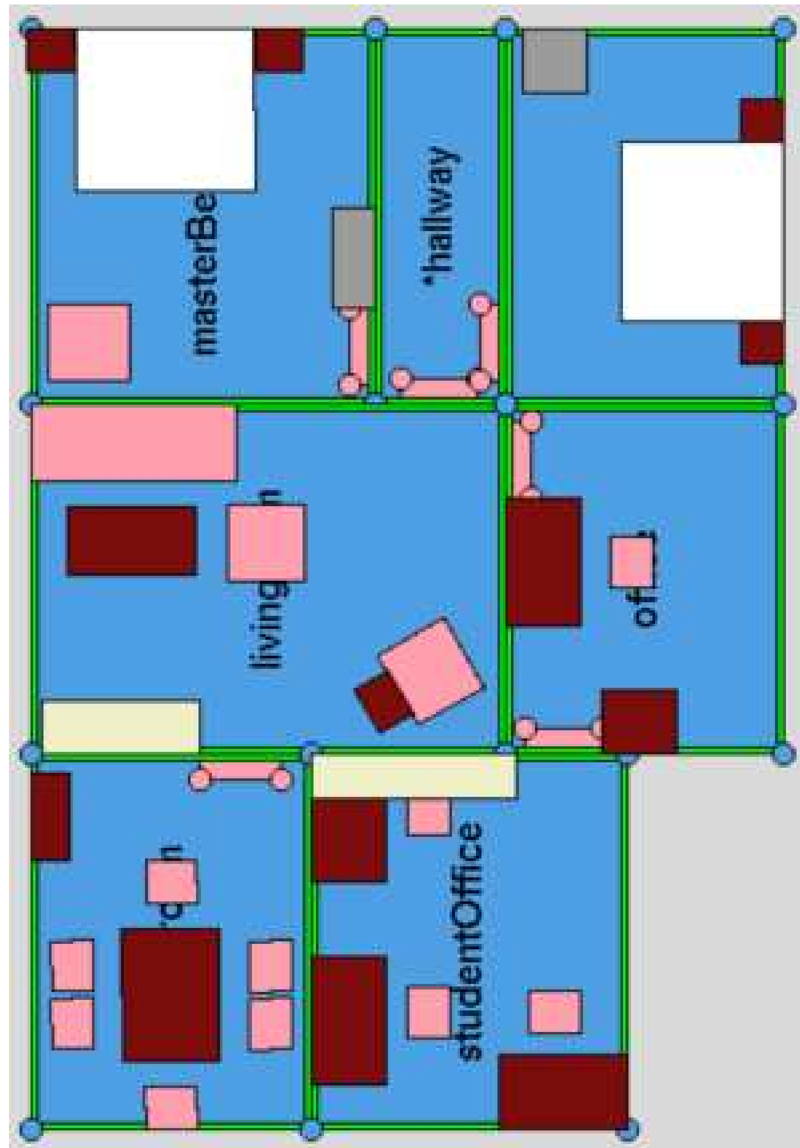


Figure 5-8: Second sample configuration.

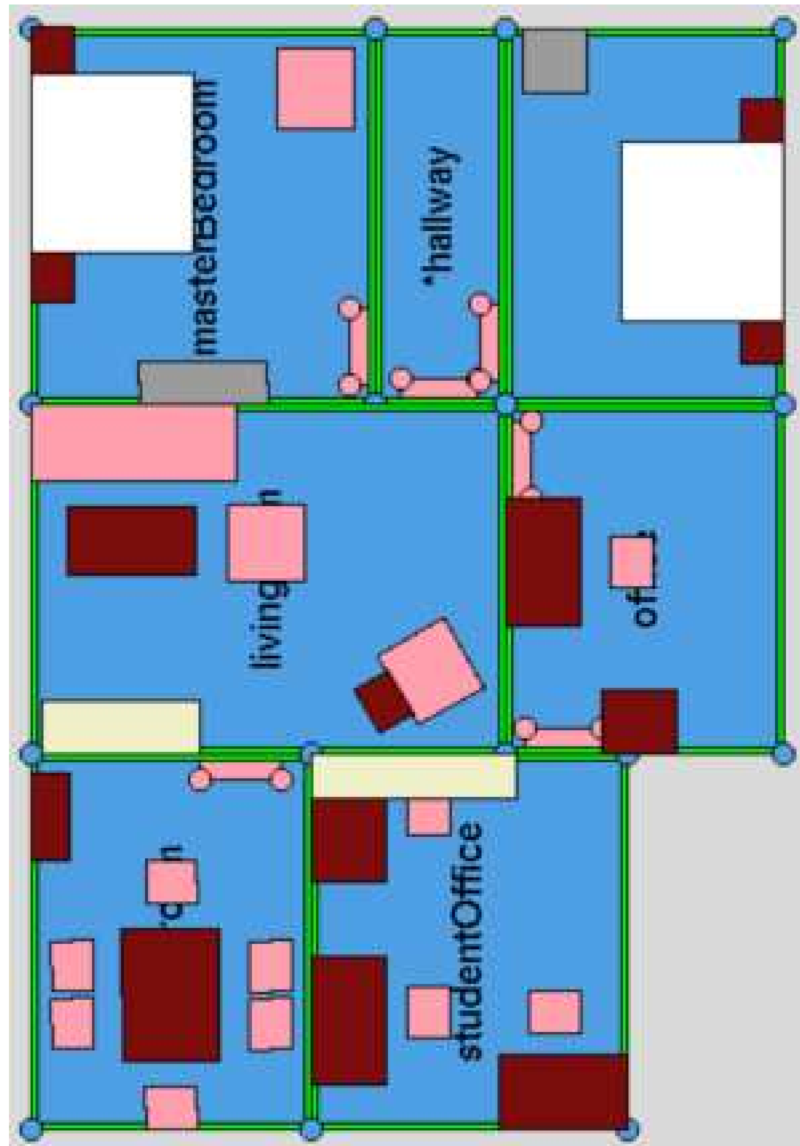


Figure 5-9: Third sample configuration.

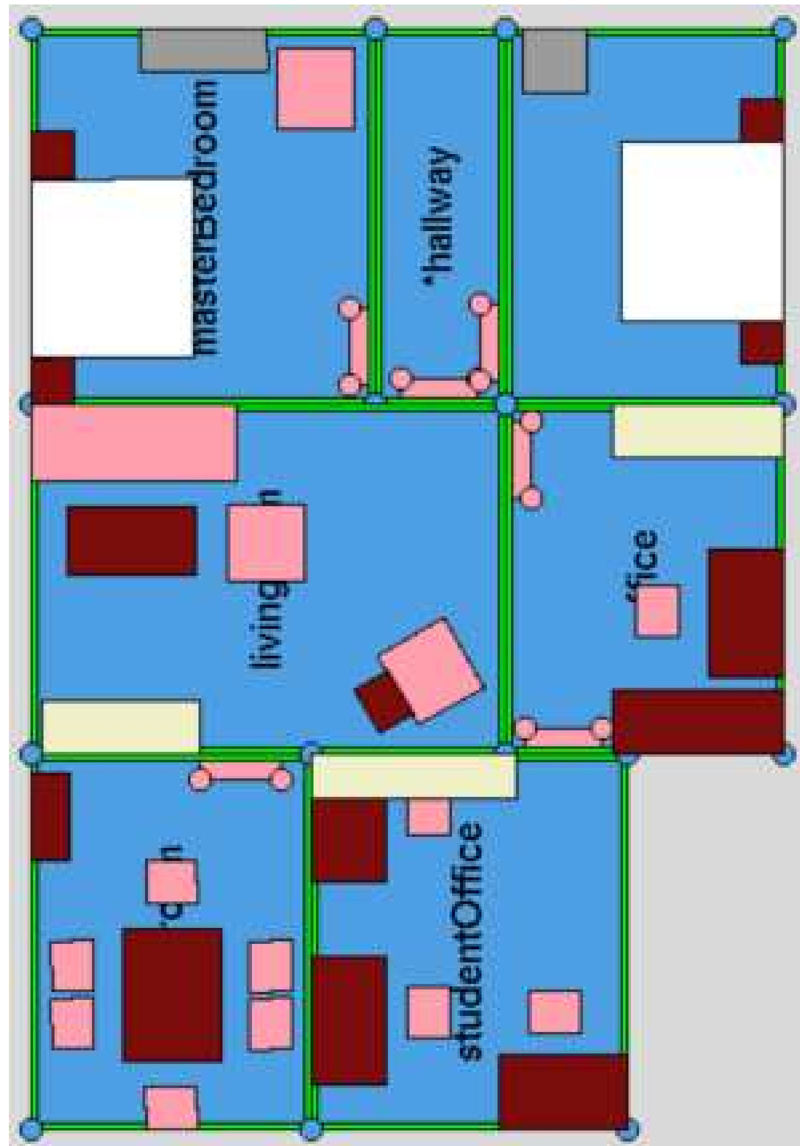


Figure 5-10: Fourth sample configuration.

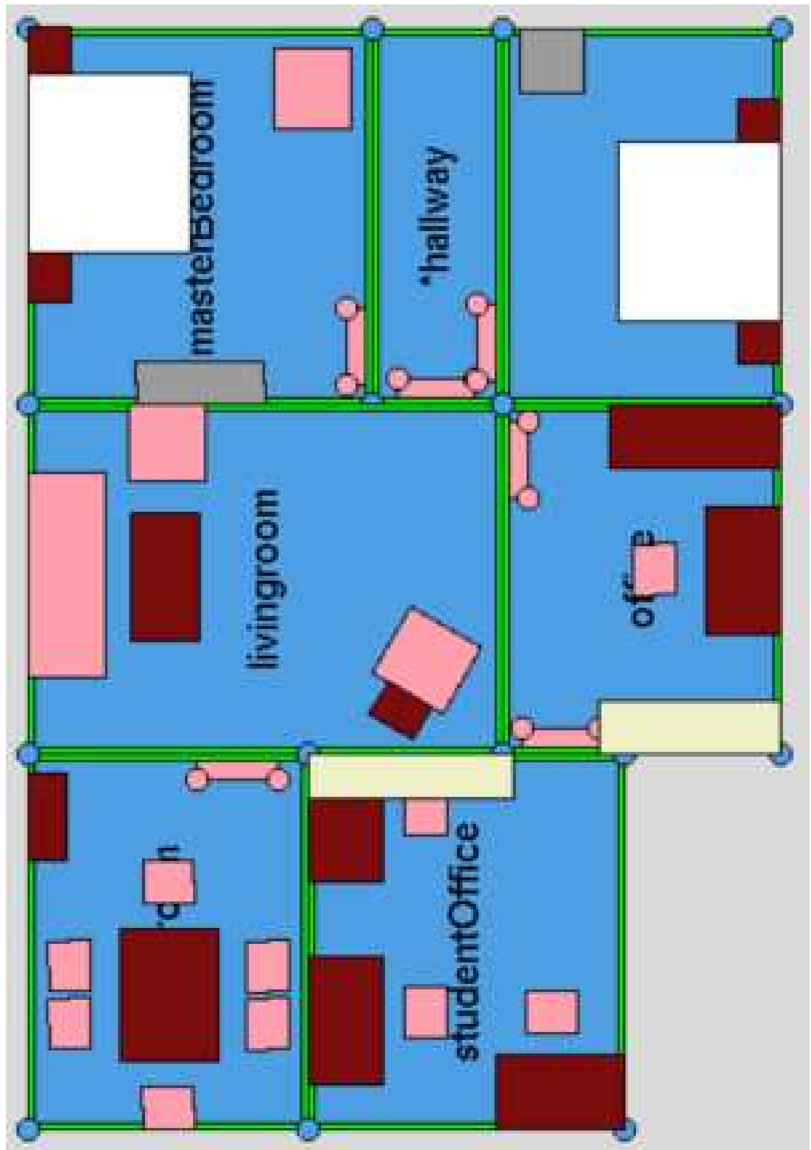


Figure 5-11: Fifth sample configuration.

Chapter 6

Discussion and Conclusion

Now that the structure and algorithms of the FurnIt system have been described, it is possible to discuss the shortcomings and successes of the FurnIt system.

6.1 Future Work

FurnIt provides a powerful framework for furnishing rooms. While the basic framework is already in place, there are several improvements that can be made to the system.

Currently FurnIt has only six default templates. These cover a wide range of room types, but a richer set of default templates should be developed. Also, the interface for creating custom templates should be improved. The FurnIt template classes provide an interface for creating new templates. However, this still requires the user to write code that links against the FurnIt libraries. This can be both cumbersome and labor intensive. A graphical user interface for specifying new templates would be ideal, since reasoning about spaces is inherently visual. One solution would be to enhance the Floorsketch interface to allow the user to draw out new templates using icons representing pieces of furniture and already defined templates. These new templates could then be added procedurally to the template type table maintained by the FurnIt libraries.

FurnIt was originally developed in order to furnish models of private homes. In spite of the addition of templates appropriate for different types of offices, the range of provided default templates reflects the initial focus on private homes. Developing new default templates for room types found in public buildings is not sufficient to make FurnIt ideal for furnishing models of these types of buildings. Building codes [20], [19], occupational safety

regulations and fire regulations [5] differ greatly between private and public buildings. If we want to create truly functional furniture configurations for public buildings, we need to create a new set of heuristics that ensure that egress, handicap access etc. are maintained in public spaces.

6.2 Conclusion

Automatic furniture population is clearly necessary when creating models of large indoor architectural environments, since the time it would take to furnish these environments by hand would be prohibitive. FurnIt provides a powerful framework for furnishing models of large buildings. Once a set of templates are created, the process is fully automatic. The given set of templates can be used to automatically furnish large numbers of buildings with no user input.

The FurnIt template model is flexible and can easily model a wide range of room types. In principle, the system can reason about an arbitrarily deep hierarchy of templates, allowing FurnIt to reason about complex furniture configurations.

Finally, a major strength of the FurnIt system is that it allows the user to compactly encode rules for how a given room type should be furnished. The given specification is inherently flexible, accommodating different room sizes and door and window configurations, resulting in a range of final configurations. Thus a building containing a large number of rooms of a given type does not require a range of templates for this room type. A single template will provide the range of configurations necessary to avoid monotony.

Appendix A

Tables

A.1 Default Room Template Types

default room types	*size (inches)*	*furniture groups in room*	*initial size*
living room	224×178	conversation reading entertainment	$132 \times 126 \times 40$ $62 \times 68 \times 52$ $37 \times 21 \times 72$
dining room	158×134	meal tablespace	$128 \times 104 \times 30$ $50 \times 18 \times 28$
master bedroom	184×160	bedset storage storage seating	$125 \times 84 \times 52$ $60 \times 50 \times 40$ $30 \times 50 \times 70$ $36 \times 50 \times 40$
bedroom	136×126	bedset storage seating	$120 \times 80 \times 52$ $60 \times 50 \times 40$ $30 \times 48 \times 40$
breakfast room	148×128	meal	$90 \times 90 \times 30$
office	148×128	workspace table shelves	$70 \times 72 \times 40$ $80 \times 30 \times 30$ $96 \times 25 \times 72$
student office	148×128	workspace $\times$ 3 shelves	$60 \times 65 \times 40$ $96 \times 21 \times 28$

Table A.1: Table of default room template types.

A.2 Default Furniture Group Template Types

default furniture group types	*min and max size (inches)*	*furniture in group*
conversation	$110 \times 80 \times 40$ $132 \times 126 \times 40$	love seat, table, chair sofa, table, three chairs
reading	$58 \times 48 \times 40$ $62 \times 68 \times 52$	chair, side table chair, side table
entertainment	$37 \times 21 \times 72$ $none \times 21 \times none$	TV with stand wall section
meal	$90 \times 90 \times 30$ $104 \times 170 \times 30$	table and chairs table and chairs
work	$50 \times 50 \times 30$ $72 \times 60 \times 30$	desk and chair desk and chair
bedset	$78 \times 78 \times 50$ $84 \times 140 \times 52$	full size bed and nightstands king size bed and nightstands

Table A.2: Table of default furniture group template types.

A.3 Default Furniture Template Types

default furniture types	*min and max size (inches)*	*furniture in template*	*func. space*
storage	$30 \times 50 \times 40$	small dresser	yes
	$60 \times 50 \times 70$	armoire	yes
seating	$30 \times 48 \times 40$	chair	yes
	$36 \times 50 \times 40$	chair	yes
tablespace	$18 \times 18 \times 24$	table	no
	$70 \times 24 \times 30$	table	no
counter	$60 \times 54 \times 36$	counter	yes
	$none \times 54 \times 36$	counter	yes
cabinets	$12 \times 42 \times 30$	row of cabinets	yes
	$none \times 42 \times 30$	row of cabinets	yes
refridgerator	$30 \times 60 \times 55$	refridgerator	yes
	$42 \times 60 \times 72$	refridgerator	yes
stove	$20 \times 24 \times 36$	stove	yes
	$46 \times 24 \times 36$	stove	yes
toilet	$36 \times 52 \times 40$	toilet (one size fits all)	yes
shower	$42 \times 36 \times 72$	shower	no
	$50 \times 50 \times 80$	shower	no

Table A.3: Table of default furniture template types.

A.4 Default Furniture Icons

default FurnIt icons	*furniture represented*	*color*
seating	chair, sofa	pink
table space	table, desk	brown
shelves	shelves	beige
beds	beds	white
storage	dresser, armoire	gray
kitchen/bathroom supplies	refridgerator, stove, toilet, shower etc.	white
clearspace	open space, used for debugging only	blue
unresolved template	used for debugging only	green

Table A.4: Table of furniture icon colors.

Appendix B

Setting Up and Running FurnIt

FurnIt is part of the Fire Walk framework. In order to run FurnIt, the user must compile the entire Fire Walk source tree.

The following are instructions for setting up and running Fire Walk. Locations of the FurnIt source files within the Fire Walk source tree are given, as well as a brief description of the function of each FurnIt source file.

B.1 Setting Up FurnIt

The Fire Walk source tree is checked in to a code repository on the MIT Graphics Group server using the cvs code versioning tool. The following are descriptions for how to check out the code, set the necessary environment variables and compile the source tree.

- Set the CVSROOT environment variable to `/d9/projects`.
- Set the BASE environment variable to the desired root directory of the source tree. The name of the Fire Walk root directory is `walkthru`. Thus, BASE should be set to `<path to where you want to store source tree>/walkthru`.
- Run `cvs checkout walkthru` in the directory where you want to store the code. This will copy the entire tree of source files into this directory.
- Go to the directory `$BASE/bin` and set the file permissions of the scripts in this directory to be world readable and executable.
- Run the script `wksetvars`. This will set up all the necessary environment variables.

- Go to `$BASE/src` and run `pmake`. This will compile all Fire Walk libraries.
- Go to `$BASE/util/firewalk` and run `gmake`. This will create the Fire Walk executable and store it in `$BASE/SYSTEM/$os/bin/`.

If the user plans to run Fire Walk on a regular basis, she should set `BASE` and source `$BASE/bin/wksetvars` in her .cshrc file.

B.2 Running FurnIt

To run FurnIt, run Firewalk and select "Run Floorplan Sketcher" from the "Subprograms" menu. This will bring up the main Floorsketch window.

B.3 FurnIt Source Files

The control functions and conflict resolution procedures of FurnIt are implemented in the `$BASE/src/FURN` library.

- `FURN.h` includes all header files. It is included from all source files in the library.
- `FURNdepend.h` includes header files of other libraries necessary for the FURN library. It is included from `FURN.h`.
- `Inference.cpp` contains setup procedures and overall control procedures.
- `Resolve.cpp` contains all conflict resolution procedures.
- `Furniture.cpp` contains all procedures for placing pieces of furniture into leaf templates.
- `Useful.cpp` implements macros and utilities functions.

The template classes are implemented as part of the `SLF` library.

- `Template.cpp` implements all template classes.
- `Furniture.cpp` implements all furniture classes.
- `Util.h` and `Util.cpp` contains the keywords for furniture and templates used by the SYLIF parser.

The Floorsketch interface (including FurnIt extensions) is implemented in the file `$BASE/util/firewalk/fgen.tcl`. The Floorsketch C++ callbacks, extrusion procedures and translators are implemented by the `$BASE/src/fgen` library. Note that any callbacks used to interact with the Floorsketch data structures must be registered in the file `$BASE/src/fgen/fgen.cpp`.

Bibliography

[1] Cristopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, NY, 1979.

[2] Cristopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, New York, NY, 1977.

[3] Richard Bukowski. The walkthru editor: Towards realistic and effective interaction with virtual building environments. Master's thesis, University of California, Berkeley, 1995.

[4] Richard Bukowski and Carlo Séquin. Interactive simulation of fire in virtual building environments. In *Proceedings of SIGGRAPH 97*, Los Angeles, CA, August 1997.

[5] City of Boston Fire Department, Boston, Massachusetts. *Fire Prevention Code*, August 1979.

[6] Alain Demachy. *Interior Architecture and Decoration*. William Morrow and Company, Inc., New York, NY, 1974.

[7] Design Basics Inc., Omaha, NE. *Heartland Home Plans*, 1995. Catalogue of floorplans.

[8] Laura Downs. Interchange format for symbolic building design. Master's thesis, University of California, Berkeley, 1999.

[9] Home Plan Design Service Inc., Omaha, NE. *Gold Seal Home Plans*, 1997. Catalogue of floorplans.

[10] Kimberle Koile. Design conversations with your computer: evaluating experiential qualities of physical form. In R. Junge, editor, *CAAD Futures 97*, pages 203–218, Boston, 1997.

[11] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1995.

[12] Leslie Linsey. *First Home: A decorating Guide and Sourcebook for the First Time Around*. William Morrow and Company, Inc., New York, NY, 1998.

[13] Julius Panero and Martin Zelnik. *Human Dimension and Interior Design: A Source Book of Design Reference Standards*. Billboard Publications, Inc., New York, NY, 1979.

[14] Anoop Parikh. *Making the Most of Small Spaces*. Rizzoli International Publications, Inc., New York, NY, 1994.

[15] Paul Raines and Jeff Tranter. *Tcl/Tk in a Nutshell*. O'Reilly, 1999.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[17] Seth Teller. Toward urban model acquisition from geo-located images. In *Proceedings of Pacific Graphics '98*, Singapore, October 1998.

[18] Seth Teller, Tom Funkhouser, Delnaz Khorramabadi, and Carlo Séquin. The ucb system for interactive visualization of large architectural models. *Presence*, 5(1):13–44, 1995.

[19] U.S. Department of Housing and Urban Development, Boston, Massachusetts. *HUD Handbook 4910.1, Minimum Property Standards for Housing*, July 1994.

[20] William Francis Galvin, Secretary of the Commonwealth, Boston, Massachusetts. *Commonwealth of Massachusetts State Building Code, 780 CMR*, November 1998.

[21] Mary Ann Young and David Nussbaum. *The Idiot's Guide to Decorating Your Home*. Alpha Books, New York, NY, 1997.