Smart Objects: Constraints and Behaviors in a Three-Dimensional Design Environment

Dustin Anthony Eggink

A thesis submitted in partial fulfillment of the requirement for the degree of

Master of Architecture

University of Washington

2001

Program Authorized to Offer Degree: Department of Architecture
University of Washington

Abstract

Smart Objects: Constraints and Behaviors in a Three-Dimensional Design Environment

Dustin Anthony Eggink

Chair of the Supervisory Committee:
Associate Professor Mark D. Gross
Department of Architecture

Each new design problem in architecture presents a new set of requirements. A designer must remain aware of these requirements and effectively communicate them to collaborators because the degree to which the requirements are met will determine the success of the solution. This thesis explores how design can be effectively presented in a medium that is both explorative of form and descriptive of the design problem's requirements. To facilitate this, we present Smart Objects, a constraint-based three-dimensional (3D) computer program. In Smart Objects, design intentions of an architectural problem are embedded as constraints into the modeled objects that compose a formal solution. A model is presented through a 3D Virtual Reality Modeling Language (VRML) viewer and constrained by a software program we wrote in the Java language. Both the VRML viewer and the Java program are contained within a single web page. In Smart Objects, as a designer meets or violates constraints, objects behave in a manner that reflects the requirements of the problem and intentions of the designer. Smart Objects communicates the design principles and guidelines that inform an architectural design to the collaborators involved in the project. It ensures that these principles and guidelines are maintained as the design progresses.
# Table of Contents

List of Figures ........................................................................................................... iii

Chapter One: Introduction

Design as Satisfaction ............................................................................................. 1
Objects, Constraints and Object Behaviors .......................................................... 1
The Design Process, Constraint Communication and Maintenance ..................... 4
Document Outline ..................................................................................................... 5

Chapter Two: Smart Objects One

The Languages that Run Smart Objects .................................................................. 7
Two Demonstrations of Smart Objects One ............................................................ 8
How Smart Objects One Works ................................................................................ 11
Shortcomings of Smart Objects One ......................................................................... 15

Chapter Three: A Background in Constraints

Computer Aided Drafting and Computer Assisted Design ................................. 16
Parametric Design .................................................................................................... 16
Constraint Programming: a brief overview ............................................................ 18
Constraint-Based Architectural Design .................................................................. 22
Constraint Considerations for Smart Objects ......................................................... 23

Chapter Four: Smart Objects Two

The Cassowary Constraint Solving Engine ............................................................ 24
The Vanna Venturi House Demonstration ............................................................... 24
Adding and Removing Constraints in Smart Objects Two .................................... 29
How Smart Objects Two Works ............................................................................ 33
How Constraints are Added and Removed on the Fly .......................................... 36
Conclusion .............................................................................................................. 36

Chapter Five: Assessment of Object Behaviors

Object Behaviors ..................................................................................................... 37
Constraint Communication ...................................................................................... 37
Designer/Model Interaction and Constraint Types ............................................... 38
The Functionality of Smart Objects ........................................................................ 40
Conclusion .............................................................................................................. 41

Bibliography ........................................................................................................... 43

Appendix A: Smart Objects One: Beam Example HTML Code ............................ 45
Appendix B: Smart Objects One: Beam Example VRML Code ............................. 46
Appendix C: Smart Objects One: Beam Example Java Code ................................ 56
Appendix D: Smart Objects One: Sun/Building Example HTML Code ................ 60
List of Figures

Figure Number                                      Page
1.1 A living-room ensemble..........................................................3
1.2 A living-room ensemble with relations diagrammed........................3
1.3 A turned couch violates the constraints .........................................3
1.4 The table is turned to re-satisfy the constraints ..............................3
2.1 VRML coordinate configuration.....................................................7
2.2 The Cosmo Player control panel....................................................8
2.3 The Smart Objects One interface...................................................9
2.4 Different perspectives of a VRML model..........................................9
2.5 Five beams spanning between two walls...........................................10
2.6 The middle beams resize for structural reasons.................................10
2.7 The end beams resize for structural reasons.....................................10
2.8 Three building masses constrained by color...................................11
2.9 Eight sets of coordinates in a VRML box.......................................12
2.10 Diagram of how the Smart Objects One applet works.........................13
2.11 Constraint calculation diagram....................................................15
3.1 Four instances of an example from Alan Boming’s Thing lab...................19
3.2 A constraint graph describing the constraint relationships in a rectangle.20
3.3 Variables L, A, H and T are assigned values...................................20
3.4 The rest of the variables are solved for using propagation....................20
3.5 A constraint graph of two related equations....................................20
3.6 Propagation fails if the wrong variables are given..............................20
3.7 Area under a curve to be solved for..............................................21
3.8 Calculating the area using linear approximation................................21
4.1 Vanna Venturi House façade.......................................................25
4.2 The two halves of the façade, divided in the middle...........................25
4.3 VRML model of Venturi House façade.............................................26
4.4 Constraint behavior of façade halves..............................................26
4.5 Chimney, stairwell and entry spatial preference................................26
4.6 Chimney, stairwell and entry spatial resolution.................................27
4.7 Stairwell becomes translucent as it is moved off center........................27
4.8 The chimney and entry change color when moved away from stairwell....27
4.9 Back of the Vanna Venturi House..................................................28
4.10 Visual balance between chimney and glass door................................28
4.11 As the designer moves the chimney right, the door moves left..............28
Chapter One: Introduction

1.1 Design as Satisfaction

Architectural design ultimately must determine the arrangement of physical forms. Forms combine and relate to one another to create the overall spaces that fulfil the classic Vitruvian elements of architecture: commodity, firmness and delight (Venturi, 1966). In other words, the objective purpose of architecture is not just the creation of space by forms, but also the fulfillment of purpose. In “Notes on the Synthesis of Form, “ (1967) Christopher Alexander states that a design problem begins “with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem.” Alexander argues that design is “not the form alone, but the ensemble comprising the form and its context.” A design solution is as much defined by the problem it solves as it is by the form it takes.

Architectural problems are complex. In “Complexity and Contradiction in Architecture” (1966) Robert Venturi argues complexity is the very nature of architecture because it includes the three Vitruvian elements. He further states that the requirements of modern structural systems and mechanical equipment make task of designing even the simplest building complicated. The task of an architect is to evaluate the complexity of a design problem and navigate towards a solution that resolves it. This thesis seeks to aid in this navigation.

1.2 Objects, Constraints and Object Behaviors

This thesis will aid a designer’s exploration towards a design resolution by representing a design problem in terms of objects, constraints and object behaviors. Although these terms may have broader definitions, in this thesis we will use them with quite specific definitions. These definitions follow.

An object is a physical form that in combination with other objects, compose a space or building. For example, a room consists of objects that are walls, ceiling, and floor. In Alexander’s terms, objects make up the “form in question” in a design problem.

Constraints are the parts that make up a design problem’s context. Generally, they are the individual requirements that a designer must address. Specifically, constraints are relationships between objects that, when satisfied, resolve a part of a design problem. For example, a design problem requires a room bounded by four walls to have a minimum area. If the walls compose a rectangular form in plan, the distances between the two pairs of opposite walls are
constrained and related because part of the design problem (minimum area) is calculated by multiplying the two distances together.

Object behaviors are changes to an object's attributes that occur when a designer moves or alters objects in ways that violate constraints. An object behavior is a response by the designer (or in the case of Smart Objects, the computer) that either indicates that a constraint has been violated, or resolves the constraint. In the case of the four walls constrained by the distances between them, if the designer moves one wall so far that the minimum area requirement is no longer met, a possible object behavior (response) would be to move the opposite wall in an equal direction, re-satisfying the constraint.

We use these three terms because we see design problems as expressions of active relationships. For example, two magnets placed close together will attract or repel one another. While the magnets (objects) themselves remain constant, the relationship (constraints) between the poles of the magnets will determine whether they attract or repel each other (object behaviors). To demonstrate this as a design process, consider the following scenario: In Figures 1-4 below, a designer is laying out furniture in a living room. The living room furniture consists of three objects: a sofa, television, and coffee table. Additionally, two constraints are given: 1) the television screen must be visible from the front of the sofa and four to ten feet away and 2) the coffee table's length must be parallel to the sofa and two to three feet away. Figure 1.1 represents one possible solution to the design problem. Figure 1.2 proves that this is a solution by overlaying a set of diagrammatic elements that represent the given constraints. As the designer turns the sofa in Figure 1.3, we see that the second constraint of the design problem (the coffee table is parallel and two to three feet away from the sofa) is violated. In Figure 1.4 the designer has invoked an object behavior and turned the coffee table, re-satisfying the constraint.
This method of design of testing, altering and re-testing is not new. It is a common design process that is extensively utilized by students and practitioners of architecture. Why, therefore, do we find it necessary to aid in this process? We find it necessary for two reasons. The first was explained by both Venturi and Alexander: the constraints that compose an architectural problem are numerous and complex. As one alters the objects of a design solution, it is impossible, for even the most agile mind, to also manage every constraint and object behavior affected by the alteration. The second reason is that architecture is a collaborative process that can involve designers, clients, consultants and contractors. These parties have a role in resolving not only a design problem's solution, but also its context. In other words, collaborators specify the constraints of a design problem. Using Vitruvius’ terminology,
collaborators determine what will make a building commodious, stable, and delightful. As a collaborator with clients, consultants and other designers, an architect must not only resolve the constraints of the design problem into a formal solution, but also be able to present that formal solution to other collaborators and demonstrate that it is an adequate resolution. We seek to aid the design process by enabling a designer to both manage the constraints related to a formal solution and provide a medium for presentation that can communicate both the form (objects) and context (constraints) of a design.

1.3 The Design Process, Constraint Communication and Maintenance
Architectural design can involve many kinds of medium. Sketches on napkins, hand line drawings on vellum, watercolor perspectives, and massing models are some of the many media used to represent the details of a design. Designers create a visual representation of physical forms and the spaces the forms combine to make. For an architect, these diverse media are a means to discover form and formal relationships that resolve the constraints of a design problem. Paul Laseau (1991) states that design is a process of visual thinking, where a design evolves by cycling from the mind, to the hand, to the medium, and back to the mind.

Architectural models are also an effective method to aid communication among collaborators because they enable a similar cycle of visual thinking—one that goes from the model, to the mind, to the discussion of the design. We have observed that a model works especially well as a design communication tool because of the level of interaction it allows. A client, collaborator or critic can move around, alter and otherwise actively engage the model. It can enable a complete cycle of visual thinking to occur.

However, drawings and models are static representations of static forms. They represent only the objects that compose a formal solution, and do not necessarily present the whole design as a solution and its problem context. Surprisingly, there are no effective means by which the design problem itself can be visually explored or communicated in a dynamic form. We seek to provide such a means. Just as a formal solution becomes more refined as it cycles through the visual thinking process, we believe that the design problem could benefit from this same refinement: the more clearly defined a problem, the more appropriate its solution will be. This is especially important in design collaboration because it is collaborators who provide the constraints to a design problem. (For example, a client who is in need of the building’s space and an engineer who specifies the building’s structural and mechanical requirements both add constraints to the design.)
The Smart Objects computer program is a means to explore and communicate the constraints that make up the context of a formal solution in a design problem. Smart Objects presents a formal design solution as a computer-generated 3D model. Like physical models, computer-mediated models in Smart Objects are interactive. Designers and collaborators can move around and within a Smart Objects model to view it from different positions. They can also move objects in the model to explore the forms and spaces they create. Going beyond a physical model though, Smart Objects presents the context's constraints by invoking object behaviors as the user meets or violates the constraints. For example, the aesthetic principle of symmetry may guide an architect. In Smart Objects, an architect could specify this constraint and dictate that if we move a column on one side of the building, a column on the opposite side must move as well to maintain symmetry.

In Smart Objects, object behaviors help designers and collaborators become as familiar with the constraints of a problem as they are with its solution because the constraints are part of the visual thinking cycle. Design moves from the mind, to the hand, to Smart Objects, which invokes an object behavior (according to the constraints), then back to the mind. Smart Objects' object behaviors enable a designer to constantly interact with and maintain constraints—whether they are derived from commodity, firmness or delight—throughout the design process.

As design collaborators actively engage a design in Smart Objects, the constraints of a problem, as well as its solution can constantly be modified to help clarify and reassess the needs of the problem and ensure that what is designed and built meets those needs.

1.4 Document Outline

This thesis describes a prototype computer program that we built to aid in the maintenance and communication of constraints in the architectural design process. We begin this discussion in the following chapter by introducing the first implementation of Smart Objects, Smart Objects One. This chapter first demonstrates the interface of Smart Objects through two architectural design scenarios. The chapter concludes with an overview of the basic computational structure that was used to build Smart Objects One, followed by a discussion of its shortcomings.

Chapter Three is an investigation into the general field of constraint-based design. We argue constraint-based design's advantages over the closely related method of parametric design, explain the basic types of current constraint solving methodologies, and review other constraint-based architectural design applications. Chapter Four addresses Smart Objects One's shortcomings by introducing Smart Objects Two. Smart Objects Two employs the Cassowary constraint-solving engine to manage constraints. The features of Smart Objects Two are demonstrated in a model of Robert Venturi's Vanna Venturi House. In this example a designer
can rework the formal arrangement of the design while Smart Objects maintains the formal relationships that guided Venturi. The second half of Chapter Four describes how Cassowary is integrated into Smart Objects Two. Chapter Five concludes this thesis with an assessment of the Smart Objects project, its achievements, shortcomings and future directions this project may take.

All versions of Smart Objects, source code and resources related to this thesis can be found at http://depts.washington.edu/smartobj.
Chapter Two: Smart Objects One

We built two implementations of Smart Objects, Smart Objects One and Two. While both versions employ the same two computing languages, VRML and Java, Smart Objects One employs only elementary constraint programming techniques and serves as a proof-of-concept demonstration to our thesis of representing design in terms of objects, constraints and object behaviors. Section 2.1 explains how Smart Objects One utilizes VRML and Java. Section 2.2 demonstrates Smart Objects One in two small design scenarios. Section 2.3 outlines the process by which the Smart Objects application works. This chapter concludes with a discussion of Smart Objects One’s shortcomings. All discussions of three-dimensional environments and models refer to the coordinate system illustrated in Figure 2.1 where X refers to width, Y to depth, and Z to height.

![Figure 2.1 VRML coordinate configuration.](image)

2.1 The Languages that Run Smart Objects

A primary goal of this project was to provide a design environment that can be accessed by all parties involved in a project. Therefore, the technology used to build the environment, (specifically, computing languages) also had to be accessible. Three components were required to implement Smart Objects: 1) a display and interaction environment for three-dimensional (3D) models, 2) a method to constrain the 3D objects in the model, and 3) a means to deliver the application to the collaborating designers.

The Internet is an effective method of delivering software applications as well as information about an architectural project to users. HyperText Markup Language (HTML) is the language used to construct and convey web pages for Internet browsing applications (e.g. Netscape's Communicator, Microsoft’s Internet Explorer). HTML is a simple language that describes the
content and to a limited extent the layout of a web page. However, an HTML page can only provide the most limited form of interaction (display of content and hyperlinks). To enable more sophisticated levels of interaction (complex animations, content posting), other languages must be employed, such as JavaScript and Perl, or the page must utilize a plug-in such as Macromedia’s Flash Player. One such plug-in supports the Virtual Reality Modeling Language (VRML). (The web browser + plug-in is referred to as a “VRML browser” – it can display 3D models described in VRML).

A VRML browser is a freely available application for viewing and interacting with 3D models. Through a control panel, one can navigate a 3D world to gain different viewpoints of objects and spaces. Figure 2.2 shows the control panel of the VRML browser, Cosmo Player (http://www.cai.com/cosmo/). It is also easy to produce VRML models. Many commercial modeling applications such as FormZ and 3D Studio Max can save 3D models as VRML “worlds.” For these reasons (and despite many known limitations) VRML is currently the de-facto standard for displaying 3D content on the Internet. With a VRML browser an architect can easily present a 3D view of a design solution to a client, consultant, or co-worker within a web page by referring to a VRML world within an HTML file.

![Figure 2.2 the Cosmo Player control panel. The toggle switches between the top and bottom control panels.](image)

### 2.2 Two Demonstrations of Smart Objects One

Basically, Smart Objects is a web page with two components: a VRML world and a Java applet, a small executable program. The VRML world contains a 3D model for the designer to explore and change; the Java applet constrains the changes to the design.

A designer launches Smart Objects One by calling up a web page with an Internet browser. The browser automatically loads a VRML plug-in into the window. At the bottom of the VRML

---

1 We recognize the ever-changing state of computational technology and realize that in time VRML will no longer be the standard by which 3D content is delivered on the Internet. For an overview of Internet-based 3D technologies available at this writing, see: http://depts.washington.edu/dmachine/Web3D/
frame a control panel allows a user to navigate around the world. This enables the users to view the model from any position or perspective. A second gray area below the VRML frame contains a dialog box (Figure 2.3). This area is the screen representation of the Smart Objects applet. The dialog box is used to indicate to the user whether constraints are being satisfied or violated.

Figure 2.3 the Smart Objects One interface.

The first example in Smart Objects One presents a room consisting of two walls turned in on their ends with five beams spanning between them (Figure 2.4). The designer can click on the near wall to drag it back and forth along the X-axis, adjusting the size of the room. However, when the designer drags the wall past the...
red line in the figure, the middle three beams increase in size, to accommodate the increased span (Figure 2.6). Because of the inward returns of the walls, the two end beams do not resize until the designer drags the wall past the second red line (Figure 2.7).

This type of object behavior (resizing the beams) has a direct effect on the composition of the model. The designer gains an immediate understanding of the consequences of moving an object. This object behavior is a way to indicate absolute constraints, design relationships that must be maintained (in this example, due to structural limitations of building components).

Constraints can also indicate design relationships that can more accurately be described as design preferences: relationships between objects that are desired, but not necessary; other arrangements of objects are allowed. The following Smart Objects One scenario demonstrates how constraints can also be used to represent design preferences.

Figure 2.8 depicts a building site with three building masses. Adjacent to the site on one side is a road and on the other, a park. A design preference for this scenario would be to minimize the amount of sunlight blocked by the building masses that would otherwise fall on the park. This involves a relationship between two distances: the horizontal distance between the park and the building and the vertical height of the building. The acceptable limit permitted by the sunlight preference constraint is displayed as a translucent yellow plane (sun plane).

The designer can move the three buildings around the site, adjusting their positions and heights. However, when the top of any building crosses the sun plane, the building turns color from blue to pink. The change in color indicates that a constraint has been violated, but the position of the building is not altered. When the designer moves the building back across the sun plane, the building returns to its original color, indicating that the constraint is no longer violated.
These examples explore two different design relationships that are both described with constraints: design absolutes (adequate beam section size) and design preferences (minimal solar obstruction). A constraint, when either violated or satisfied, alters the attributes of an object. We call this alteration “object behavior.” The type of object behavior affected by a constraint depends on the nature of the design relationship being indicated. For design absolutes, we use behaviors that affect the function of the object. For example, the structural strength of a beam depends on its sectional size. We therefore use object size as the behavior to indicate the design absolute in the first example. For design preferences, we use behaviors that are not essential to the object’s function. In the second example, object color is used because it is not essential to the study of building masses and formal composition.

2.3 How Smart Objects One Works

A Smart Objects One applet constrains a design by observing changes to specific parts of a model and testing them, using conditional “if, then” statements. The specific parts of a model we refer to are object nodes. A node is a set of values that the VRML browser uses to render the 3D world. For example, every geometric object in a VRML world has a node that describes its vertices, or points. Each point contains three values: its X, Y and Z coordinates in the 3D space. Therefore, an object the shape of a rectangular box would have a node that contains a set of eight X, Y, Z values (Figure 2.9).
Figure 2.9 A VRML node describing this box would contain all these point coordinates.

The following describes how Smart Objects One works². Figure 2.10 diagrams this process. The left side of the diagram illustrates the procedure, or steps, the applet follows. The right side illustrates the information in the VRML file used by the applet's steps. When a Smart Objects page is opened, the web browser loads a VRML world and a Java applet. The Java applet will monitor, or observe, specific (pre-programmed) changes in the VRML world. As the applet starts, it looks for the VRML world it will observe (step 1). The applet then locates a series of nodes in the VRML world (step 2). These nodes represent the attributes of an object the designer has specified to be constrained.

² The Java code of both Smart Objects One example applets, along with the accompanying VRML files, can be found in the appendix and http://depts.washington.edu/smartobjects.
Figure 2.10 how the Smart Objects One applet constrains objects in a VRML world

In the “three building” (solar obstruction) example described in the previous section, the applet looks for nine nodes. In order for the applet to find a node, it must be named. In these
examples, we added a name to every node the applet would look for in the VRML code. The first three nodes we'll call “Building One Movement,” “Building Two Movement” and “Building Three Movement.” These nodes contain the X, Y, and Z distances that each of the building masses have been moved by the designer from their original positions. The next three nodes are “Building One Color,” etc. These nodes contain the Red, Green and Blue (R, G, B) values of each building's color. The last three nodes are “Building One Active,” etc. These values contain the value of either “True” or “False,” depending on whether the designer is currently moving the building. The node value is “True” when the building is being dragged, “False” when it is not. Once the applet finds all these nodes, the designer can begin interacting with the objects.

The third step (3) that the applet performs is to test the value returned by the Callback function. A function named “Callback” is always running in the applet. Within Callback there are three conditional (“if, then”) statements. The first statement directs that “if” the node “Building One Active” is “true,” “then” call the function “Constrain Building One.” That is, when the designer is dragging Building One with the mouse, then Smart Objects will run the function “Constrain Building One.” The second and third conditional statements do the same for the nodes “Building Two/Three Active” and call functions “Constrain Building Two/Three.”

Each “Constrain Building” function (steps 4a, b, c) takes the values from the “Building Movement” nodes as its argument. These values are the amount the designer has moved the building mass in the X, Y and Z directions. To determine whether the building the designer is moving intersects the yellow plane, an arithmetic inequality, X – Z > 2.5, is tested as the condition of the “if, then, else” statement. 2.5 is a unique number to this example. It is the combined amount of movement in the X and Z directions that would cause the building to cross the sun plane. Figure 2.11 diagrams this logic. If the condition is true, then the R, G, B values of the node “Building One Color” are changed to produce pink. Otherwise (if the condition is false) the else clause returns the R, G, B values to their original values. As the Callback function runs, it will keep calling the Constrain Building functions, providing them with new X, Y, Z values, as long as the designer is moving the object.
2.4 Shortcomings of Smart Objects One

Smart Objects One is a "proof-of-concept" demonstration program; it works well enough to express the idea of constraints and object behaviors. However, the method used to implement constraints - the "if, then" conditionals - is a basic form of constraint programming. To use this method in a design application would present two major problems: 1) the code proliferates as the designer adds constraints and objects to the model and the designer cannot add or remove constraints while Smart Objects is running.

Every constraint applied to an object requires its own "Constraint" function the designer must code (a dubious prospect in itself) into Smart Objects One. Each constrained object requires another "if, then" statement to be added to the Callback function and a new set of nodes to be found when the applet starts. Constraints in this implementation are pre-compiled functions. They cannot be added or removed while the designer is running Smart Objects.

Even a simple building design could have over a hundred objects and potentially as many or more constraints. The limitations of this first implementation warranted an investigation into a more flexible and powerful implementation of Smart Objects. As we search to resolve the shortcomings of Smart Objects One, we will begin with a background investigation of constraint programming and constraint-based architectural design in the following chapter.
Chapter Three: A Background in Constraints

3.1 Computer Aided Drafting and Computer Assisted Design

In his article “Parametric Design. A review and some experiences,” (1997) Monedero states that architectural design applications can be classified as either assisting in the production of drawings or assisting in the design process. Tools of the former type are typically known as computer aided drafting (CAD) tools. These programs aid drawing production by automating tasks such as layers, zooming in and out, and three-dimensional extrusion. While eliminating many of hand-drafting’s repetitious actions, these tools do not actively assist a designer’s navigation towards a problem solution. Monedero argues design interaction between the computer and the designer is the principal element missing in the majority of current CAD programs and three-dimensional modeling tools. This is a limitation to assisting the design process because, as Monedero states, a fundamental aspect of design is a process, similar to Laseau’s term “visual cycling,” of going “forwards and backwards, re-labouring once and again” a design solution.

This chapter reviews computational technologies used to enable design interactivity in drafting and modeling programs. This discussion begins with an introduction and comparison between parametric and constraint-based design. Next is a review of the aspects of various constraint-solving methodologies, followed by an overview of what solving techniques are used in other constraint-based architectural design applications. This chapter concludes by outlining the solving features that we consider important for Smart Objects.

3.2 Parametric Design

As a broad classification of parametric design tools, Monedero offers the following: “forms are created by combining basic entities (primitives) that are inserted in the model after a basic template, which includes their ‘proper parameters,’ is filled. A line, for example, is an entity that becomes part of a model once two parameters, length and direction, are specified.” Although using this definition, all CAD programs can claim to be parametric, Monedero asserts that this is not true parametric design, because parametric design specifically defines relationships among groups of primitives, or families. Gross (1989) offers a specific definition of parametric design tools as those that allow “the architect to describe a family of designs which vary according to certain key variables, or parameters. A simple example is a parametric model for a stair, with height between floors and riser height as inputs.” Parametric design tools specify a directional relationship between variables. (Some variables are selected as the “design” or “driving” variables while others are the “driven” or “determined” variables.) In
Gross’s example of the stair, the tread depth, riser number, and total run distance are variables dependent on the designer’s input of floor-to-floor and riser height.

Many commercial CAD programs allow this type of parametric design through a programming language such as AutoCad’s AutoLisp. However, the most advanced architectural parametric design tool is Revit. There are two key attributes that distinguish Revit from other architectural design programs: it’s “one building, one file” information database, and it’s “parametric change engine.”

Unlike most drafting programs e.g., AutoCad or Microstation, that allow some degree of three-dimensional modeling and parameterization to be added to a conventionally constructed CAD model, Revit is inherently a parametric modeler.

From a user’s perspective, the difference between drafting (AutoCad, Microstation) and modeling (Revit, FormZ) programs is that in a drafting program, a building is designed through several two-dimensional (2D) drawings that depict different parts of the building (sections, plans, elevations). In a modeler, a building is designed as a three-dimensional (3D) composition, from which architectural drawings (sections, plans and elevations) are rendered as views of the same model. This difference is important because if a change is made in a drawing, it occurs in only one of several 2D drawings that represent the building. This can result in inconsistencies.

For example, a room designed in a drafting program consists of five drawing files: a floor plan and four separate interior elevations for each wall. When a change is made to the design, (one wall is moved two feet) the designer must individually alter every drawing the change affects (three of the wall elevations and the floor plan) because each drawing is a separate file. This can be a critical issue in parametric design because a single change can cause changes to ripple throughout the design. (For example, a designer moves two supporting columns further apart in plan view, which changes the section view and increases the size of the supportive beam.)

When a parametric change is made in a model, it occurs in one 3D model that represents the building, maintaining consistency. Revit is the first, and only, commercial parametric modeler that is intended for architectural design.

Parametric design is appropriate for routine design tasks, such as structural design, where entering variables into a well-defined formula can quickly produce a series of design alternatives. Gross argues, though, that this is not a suitable approach to conceptual design because an architect may not want to determine directional relationships in this explorative
phase. He argues that a more appropriate method for explorative design is to define the
relationships between elements, but not determine a direction. That is, constraint-based design
is more appropriate for conceptual design.

3.3 Constraint Programming: A brief overview

describes constraints as “a logical relation among several unknowns (or variables), each taking
a value in a given domain. The constraint thus restricts the possible values that variable can
take, it represents partial information about the variables of interest.” Constraints are common
in everyday life and provide a guide for our reasoning. Personal time management is a typical
use of constraints. On any given day, we set a number of tasks to accomplish (work, errands,
leisure, meetings, eating, and rest) within a limited amount of time (a day). To meet these
goals, we rank the tasks in order of importance, limit the amount of time we spend on each
task, see what tasks can be performed at the same time and schedule our day to minimize the
time spent going from one task to the next. Typically, we work to solve not one but many
related constraints, which characterizes the task domain of constraint programming. Bartak
describes it as follows:

Constraint programming is the study of computational systems based on constraints.
The idea of constraint programming is to solve problems by stating constraints
(requirements) about the problem area and, consequently, finding (a) solution
satisfying all the constraints.

Constraint programming has been the subject of intermittent research since the 1960’s. Over the
last ten years, interest in the field has increased because of its potential to solve difficult real-
life problems. In addition to architecture, scheduling, database management and circuit design,
have benefited from advances in constraint programming (Bartak, 1999).

The Sketchpad program, introduced by Ivan Sutherland in 1963, was the first application of
constraints in a computational environment (1963). This system allowed users to create and
constrain points, lines and arcs. For example, two lines could be constrained to be parallel.
When the user manipulated these items, Sketchpad would analyze the changes and adjust the
geometries so that all the specified constraints were met. A successor to Sketchpad, ThincLab,
written by Alan Boming (1981) allowed users to manipulate constrained geometric figures.
Figure 3.1 depicts an example of ThincLab where a parallelogram’s vertices are constrained to
the midpoints of a quadrilateral. The user can alter any vertex’s position (of either the
parallelogram or the quadrilateral) and ThincLab will maintain the specified constraints. For
designers these two applications are significant because of their graphic interface. Bartak
claims that they are also significant because of their contributions to the development of local propagation, a computational search method for finding constraint solutions.

Figure 3.1 four snapshots of an example from Alan Borning’s Thinglab. The vertices of the parallelogram are constrained to be on the midpoints of the sides of the quadrilateral.

The defining aspect of constraint programming is that the user states what is to be solved instead of how to solve it. The question of how to solve constraints is left to the computer. Bartak categorizes constraint programming as using either satisfaction or solving techniques. Constraint satisfaction deals with problems where there are: 1) a set of variables to be solved for, 2) a finite set of values for each variable (its domain) and 3) “a set of constraints restricting the values that the variables can simultaneously take.” Using constraint satisfaction techniques, a program will find a solution that satisfies all the constraints by systematically searching through all the variables’ domain sets. This is accomplished through what is called the “generate and test” method; the application generates a set of values for the variables and tests whether they satisfy the constraints. This method works well in scenarios with few constraints and small domain sets. However, in more complex problems, generate and test constraint satisfaction can be computationally slow because it may have to run many cycles before finding a solution that satisfies all the constraints. Constraint solvers avoid this by using solving techniques based on mathematical algorithms to return correct constrained values to the application. There are many mathematical techniques that have been used to solve constraints; each one with their own strengths and weaknesses. Here, we will introduce three: propagation, simultaneous solution and relaxation.

Propagation is a process of replacing the variables \((X, Y, Z)\) of a constraint equation with constant values in order to solve for the other variables \((A, B, C)\) in the equation. For example, a rectangle has seven variable attributes: left border (coordinate position) \((L)\), right border \((R)\), top border \((T)\), bottom border \((B)\), width \((W)\), length \((L)\) and area \((A)\). We can describe the rectangle in the constraint graph below (Figure 3.2) or as three equations: \(L + R = W\), \(T + B = H\) and \(W \times H = A\). Using propagation techniques, we can substitute some of these variables with values to solve for the rest of the variables. In Figure 3.3, the variables \(L, A, H\) and \(B\) have been assigned the values 5, 20, 4 and 6. Figure 3.4 shows how the values are propagated through the constraint graph to solve for variables \(R, W\) and \(T\).
Propagation works well when enough variables can be replaced with constant values. When there are not enough input values though, propagation fails. For example, the constraint graph below (Figure 3.5) describes the two equations $X + Y = A$ and $2X + 3Y = B$. If we only input values for variables $A$ and $B$, it is impossible to solve for $X$ and $Y$. Propagation will halt because there are not enough input values to compute (propagate) an output value (Figure 3.6).

However, we can solve this constraint network using simultaneous solving. Simultaneous solving is a process of eliminating variables, for example, by substitution. In the example above ($X + Y = A$ and $2X + 3Y = B$), we can first solve for $Y$ in one equation ($Y = A - X$), substitute it for $Y$ in the second equation ($2X + A - X = B$), and then solve for $X$, ($X = B - A$). Now if we are given the values for $A$ and $B$, we can solve for $X$ and $Y$. Simultaneous solving will always produce the optimal solution (minimum or maximum) if the constraints are linear (variables are not raised to a power or multiplied by each other). If constraints are non-linear though, simultaneous solving will not always provide an optimal solution.
Thus far, we have discussed techniques for solving constraint networks that are “well-constrained”, which is to say that there are enough variable inputs and all constraints can be resolved in a single solution. However, sometimes two or more constraint equations that contradict each other can be written into the same network. This can be likened to the everyday situation of needing to be in two places at one time: there is no solution that can solve every constraint. There are two principal ways constraint solvers resolve this problem: constraint hierarchies and partial constraint satisfaction. In constraint hierarchies, preferred, or strong, constraints are solved before weaker ones. Partial constraint satisfaction weakens all or non-preferred constraints by gradually increasing their domain area, until a total satisfying solution is found.

Similar to partial constraint satisfaction, constraint relaxation is a third solving technique that is particularly valuable for solving constraint conflicts and high-powered constraint networks (many variables raised to powers). Constraint relaxation produces three things: a near solution, a sum of how far off the near solution is from the perfect solution, and a direction (positive or negative) to move towards the perfect solution. Relaxation works well in over-constrained (conflicting) networks because it does not try to resolve contradictions. Instead it identifies how far from perfect its solution is, and which direction to move towards the optimal one. A popular computational method in constraint relaxation is to use linear approximation to arrive at solutions that are near the optimum. For example, if we want to constrain the area below the curve in Figure 3.7, we could calculate the complex equation that determines the arc of the curve, or add up the area of a series of boxes drawn under the curve (Figure 3.8). The second method will not produce an exact answer, but if we are calculating a room’s maximum occupancy, for example, the approximation is close enough.

![Figure 3.7 the area under a curve to be solved for.](image)

![Figure 3.8 using linear approximation to calculate area will produce a near solution.](image)
3.4 Constraint-based architectural design

The best type of constraint solving technique depends on the situation for which it is being used. In an architectural design there may be thousands of constraint relationships in a single building design. These relationships can be complex, contradictory, linear and non-linear. Below we will overview a few constraint-based design applications using varying solving techniques to resolve different parts of the design process.

Gross’s “Co-Drow” (1989) uses propagation and simultaneous solving techniques. Propagation is used to solve linear equations, and generally “well-constrained” networks. For example, to specify that the X position of two columns (c1 and c2) are always thirty feet apart, we write the linear equation Xc1 = Xc2 + 30, handled easily through propagation. Symbolic solving enables Co-Drow to help break propagation cycles. Co-Drow allows a range of values (minimum and maximum) associated to an object to be constrained as a set and propagated.

Kolarovic’s “ReDrow” (1997) utilizes constraints to explore geometry, shape and dimension organizations in a design. ReDrow (Relational Drawing) is akin to the hand-drafting process in which a designer creates construction lines on paper to guide the final inking of a drawing. ReDrow however, takes advantage of computational power and provides a dynamic drawing environment where relationships among the construction lines can be designated. ReDrow uses constraint propagation to implement changes in a drawing by either translating or rotating objects. Constraint conflicts are resolved one of two ways: 1) in an inactive mode the constraint is eliminated, 2) in an active mode it establishes a new uni-directional relationship (or parameter, as opposed to a bi-directional constraint) between the elements.

Arvin and House (1999) express constraints on a design as forces acting upon masses. In a process they call “responsive designing,” the programmatic objectives of a design are modeled as objects and forces. Responsive designing involves physically based modeling techniques that “attempts to represent dynamic motion and changes in geometry by modeling objects as mechanical elements that behave according to the laws of physics.” To demonstrate their responsive designing method, space planning is used as an example application. Spaces are modeled as masses, and adjacencies between the spaces are modeled as springs that connect the masses. The design process begins when the user adds programmatic objectives to the design, modeled as forces applied to the masses. The system of masses and springs resolve the objective by reaching equilibrium, a process of constraint relaxation. The variable strength of a spring is a way to specify the importance of a constraint (constraint hierarchy). The designer interacts with the composition by adding, removing and altering objectives in the form of forces acting upon the masses and springs.
3.5 Constraint Considerations for Smart Objects

In Smart Objects, we sought a constraint-solving technique that could handle a large amount of constraints, able to add and remove constraints at run-time, and fast enough to run as a seamless background engine. Computational compatibility was also a consideration. Particularly, an engine built in the Java language would provide the easiest way to interface with a VRML 3D viewer. The following chapter describes Smart Objects Two, which employs the Cassowary solver, a constraint-solving engine developed by Greg Badros and Alan Bornig.
Chapter Four: Smart Objects Two

In Chapter Two we described Smart Objects One as an adequate “proof-of-concept” demonstration, but not sophisticated enough to 1) efficiently handle many constraints and 2) unable to add and remove constraints at run-time. In this chapter, we present Smart Objects Two, which resolves the shortcomings of Smart Objects One by incorporating the Cassowary Constraint Solving Engine (Cassowary). In this chapter, we briefly introduce Cassowary before describing Smart Objects Two. To demonstrate Smart Objects Two, we have modeled Robert Venturi’s Vanna Venturi House, embedded with constraints that represent the architect’s design decisions. This chapter concludes with a summary of how Smart Objects Two works.

4.1 The Cassowary Constraint Solving Engine

Developed by Greg Badros and Alan Borning (1999), Cassowary uses a simplex algorithm as the basis for resolving constraints (defined as linear equations or inequalities) through propagation. Constraint hierarchies, designated by “strength,” are used to resolve constraint conflicts. Cassowary uses a metric comparator to negotiate between different solution sets, which determines how nearly every constraint is to be satisfied.

We chose to build Smart Objects Two using Cassowary as a platform for the following four reasons: 1) it manages and solves constraints incrementally, which enables to add and remove constraints at run-time. 2) It is powerful. A building design can have hundreds of constraints; Cassowary can manage all these relationships, without any noticeable drop in performance. 3) It uses propagation to resolve constraints. Smart Objects models are constrained by position. Users interact with a model by moving individual objects, extending a value to be propagated to the constraint network. 4) Cassowary is written in Java, the same language as Smart Objects One.

4.2 The Vanna Venturi House Demonstration

In his book “Complexity and Contradiction in Architecture,” (1964) Robert Venturi explains the design relationships that influenced his design of the Vanna Venturi House (Figure 4.1). According to Venturi, symmetry, balance and scale were among the fundamental aesthetic relationships that dictated the final form of the house. In this demonstration, Venturi’s design intentions have been interpreted as constraints that Smart Objects maintains as a user (designer or collaborator) manipulates a model of the house.
As with Smart Objects One, a user launches Smart Objects Two by calling a web page. The VRML world presents an interactive model of the Vanna Venturi House. The user can move elements of the house to rework Venturi's design. Constraints and object behaviors are applied as the user interacts with the model. The following are some of the constraints the Smart Objects applet applies as interpretations of Venturi's design intentions:

1. Venturi states that symmetry and balance were carefully considered while composing the front façade. The front façade is composed of two halves (Figures 4.1 and 4.2). In Smart Objects Two, as the user moves one half of the front façade, the other half moves an equal distance in the opposite direction (Figures 4.3 and 4.4).
2. Again, concerning balance, Venturi wanted the chimney, stairwell and entry to be placed along the centerline of the house. However, all three elements could not occupy the same space (Figure 4.5). Venturi resolved this constraint by keeping the stairwell centered and placing the chimney and entrance spaces slightly off of it (Figure 4.6). We interpret this design decision with two constraints. Figure 4.7 shows the stairwell becoming more translucent as the user moves it off center. The colors of the chimney and entrance are constrained according to their relationship to the stairwell: as the chimney and entry space are moved within the space occupied by the stairwell, these elements take on the stairwell’s color, indicating a constraint violation (Figure 4.8).
3. In the back of the house, Venturi positioned a glass door off-center to visually balance the chimney (Figures 4.9 and 4.10). In this demonstration, as the chimney
is moved, the door moves in the opposite direction to maintain the intended visual balance (Figures 4.11 and 4.12).

4. Venturi states that the size of the fireplace is too big relative to the living room's size (Figure 4.13). We interpret this as a proportional relationship between the height of the chimney and the volume of the living room (Figure 4.14). In Smart Objects, as the chimney is raised, the living room gets bigger (Figures 4.15 and 4.16).
These are the four main constraints exhibited in this demonstration. Other constraints programmed into the applet are similar in nature and exhibit the same types of object behaviors.

4.3 Adding and Removing Constraints in Smart Objects Two

The second Smart Objects implementation differs from the first in that the user can add and remove constraints while running the program. This function is accomplished through the panel of buttons and the plan view rendered in the Java window (Figure 4.17). As the user moves objects in the 3D VRML world, these changes to the model are reflected in the plan view. The designer can select an object by clicking on it in the plan view. To apply a constraint, the designer selects two objects and specifies a desired relation between them. (More than two objects can be selected; however, a constraint will only be applied to the two most recently selected objects.) Smart Objects indicates the most recent object selected by coloring it red. The object selected first is colored green. A user can add and remove constraints that will set the X, Y or Z translation value of one object equal to another’s X, Y or Z translation value. Shown in Figure 4.18, the types of constraints that can be added in Smart Objects Two are: “X=Z,” “X=Y,” “X=X,” and “Y=Y.” For example, when the constraint “X=Y” is applied to object A and B, when object A is moved five feet in the X direction, object B will move five feet in the Y direction. To remove a constraint, the user selects the two objects constrained together and clicks on the “Remove Constraint’s” button (Figure 4.18). Finally, the dialog box (Figure 4.18) indicates which object, by number, has been selected, and when constraints have been successfully added or removed.

---

3 Currently the Smart Objects applet draws the plan view using only the first four X and Y coordinates of each VRML object. This technique works well for regular, box-shaped objects (objects with eight vertices), which compose most of the model. However, non-regular objects drawn with this technique results in some inconsistencies.
To demonstrate the process of adding and removing constraints, we present the following scenario: Figure 4.19 presents the Vanna Venturi House modeled in Smart Objects Two. On the right, a plan of the house is displayed in the Java applet window. In Figure 4.20, a user has changed the design by moving the façade walls further apart. This change is reflected in the plan window. However, at this stage the end walls of the house are separate elements and they do not move with the façade walls.

Figure 4.17 Smart Objects Two interface: the floor plan in the lower left, the constraint buttons and dialog box in the upper right.

Figure 4.18 Smart Objects Two’s buttons for adding and removing constraints and dialog box.

Figure 4.19 the Vanna Venturi House model (left) and the live plan view in the Smart Objects Two Java window.
Figure 4.20 the end wall and left façade wall are not aligned.

In figure 4.21, the user selects the left half of the façade wall and the left end wall and selects the constraint button “X=X.” The dialog box indicates that the constraint has been added.

Figure 4.21 the “X=X” constraint has been added to selected left façade wall and left end wall. The dialog box indicates the added constraint.

Now as the user drags the left façade wall, the left end wall moves the same amount (Figure 4.22). However, this action reveals another constraint: as the left end wall moves, the right end wall moves as well. The two end walls are constrained to move together.

Figure 4.22 the left end wall now moves with the left façade wall (left). The right end wall’s movement with the left reveals a previously programmed constraint (middle and right).

The user resolves this by selecting the two end walls in the plan view and clicking the “Remove C(onstraint)’s” button in the applet window (Figure 4.23). The dialog box indicates the constraints have been removed.
Figure 4.23 the user selects the two end walls and removes constraints.

Now as the user moves the left façade wall, the left end wall moves while the right one does not (Figure 4.24).

Figure 4.24 the left end wall’s movement is the same as the façade wall’s, and independent of the right end wall.

Finally the user selects the right façade wall and right end wall and the right façade wall and adds the constraint “X=X” (Figure 4.25). Now as the user moves a façade wall, both end walls move to maintain an alignment (Figure 4.26).

Figure 4.25 the “X=X” constraint is added to the right end and façade walls.
Figure 4.26 Both end walls are constrained to move with their respective façade walls.

4.4 How Smart Objects Two Works

This section describes how the Smart Objects Two applet uses Cassowary to implement and manage constraints. Figure 4.27 diagrams the process; the Smart Objects applet is depicted on the left, the VRML file in the upper right. There are also three components to the Smart Objects Two applet: the Cassowary solver, the “CasSet” array and the live plan view.

Smart Objects Two is opened the same way as One: the designer opens the Smart Objects web page which loads a VRML world and the Smart Objects Java applet. Smart Objects first calls another applet, Cassowary, which the user never sees or interacts with directly. Smart Objects will use the solver to define, manage, add, and remove the constraints. Next Smart Objects creates an array of objects that we’ll call CasSet. CasSet stores the nodes of VRML objects. It will contain the translation, color, opacity, and point vertex values of the constrained objects. Before storing this information the applet must first locate the nodes of the objects.

---

4 An annotated copy of the Smart Objects Two Java code, along with the VRML file of the Vanna Venturi house model can be found in the appendix.

5 Cassowary was initially built for 2D graphics management, and therefore only had X and Y coordinates available. For Smart Objects we altered the Cassowary to accommodate three dimensions by adding a Z coordinate. The changes to the Cassowary Java code can be found in the appendix.

6 Arrays are Java’s means of organizing information by allocating a sequence of slots to hold a set of data. For example, we might define an array to store information about a set of books, indexed by their ISBN. By calling a certain ISBN, one would now have access to a set of data about that book (e.g. title, author, year and publisher).
One of Smart Objects One's problems was that for every object to be added to the design in the VRML world, the programmer would have to add more lines of code to find that object's nodes. Smart Objects Two resolves this problem with three steps: first, in the VRML file, all objects
are numbered (object1, object2, etc.). Second, the nodes of each object are named according to their node type and object number (the color node of object 1 is named "color1," the translation node of object 12 is named "move12"). Smart Objects finds the nodes that describe the properties of the VRML objects by looping through the objects (from 1 to the number of objects) and retrieving the nodes for each number.

Next, the applet adds the hard-coded constraints to the solver. Using Cassowary to manage and resolve them, constraints are now defined as linear equations or inequalities instead of "if, then" statements. Cassowary only accepts linear equations or inequalities written as 0 (=, >, <) A - B. Constraints can use a constant value in place of one of the variables. For example, if "ObjectZ" represents the Z translation value of an object, \( 0 > \text{ObjectZ} - 6 \), would define a constraint that would prohibit the "Object" from moving more than six VRML units up. Variations to this formula can be performed with a linear expression. For example, the constraint between façade walls of the Venturi House, where "Right Wall X" and Left Wall X" represent the translation values of each half in the X direction, is expressed \(( - \text{Right Wall X}) \). This constraint is written as, \( 0 = \text{Left Wall X} - ( - (\text{Right Wall X})) \), which is equivalent to Left Wall X = - Right Wall X.

Constraints can either be hard-coded into the applet or added at runtime. In the Vanna Venturi example, we used the former method. The latter process will be described in the next section.

To hard-code a constraint into the Smart Objects applet, the programmer must first write a linear equation or inequality and then add it to Cassowary. The solver will manage all the values of every object according to the constraints it has been given, trying to keep the values consistent with all the constraints. Before a constraint is added though, Smart Objects gives it a unique name, which is used to identify the constraint to the designer. The name is also used to remove a constraint from the solver. By default, constraints are named according to the object numbers in the equation or inequality. Therefore, the façade wall constraint would be named "equation12." Once the constraints have been added, Smart Objects is ready for the designer to manipulate the model.

In Smart Objects One, the callback function reported which object was being moved by the designer and passed the object’s translation values to the “Constrain” functions. In this version, the callback function does the same, except, in place of a series of if-then conditionals it sends the values to Cassowary. Once Cassowary has received the new translation values, it resolves every constraint and updates the constrained values in the CasSet array. Smart Objects
then reads the values from the CasSet array and passes them to the VRML nodes, causing the model to exhibit the constrained behavior.

4.5 Adding and Removing Constraints on the Fly
Enabling constraints to be added or removed on the fly was a matter of enabling the designer to select objects through the Smart Objects interface. For this purpose, the “Draw Plan” function renders a live view of the model’s floor plan. This function adds the X and Y values from an object’s point nodes to the translation value of the same object and draws a line to the next point. Thus, the function constantly updates the floor plan to reflect the designer’s moves in the VRML world. When the designer clicks on an object in the Java applet floor plan view, it is highlighted, and the applet stores that object’s number. The applet stores the two most recent object numbers the designer has selected. When the designer clicks on one of the constraint buttons, for example “X=Y,” Smart Objects adds a constraint that specifies the X translation of the first object is to be kept equal to the Y translation of the second.

Removing constraints requires the same process of selecting objects from the floor plan view. When the “Remove C(onstraint)s” button is pushed, the solver is called to remove all constraints between the two objects selected.

4.6 Conclusion
Smart Objects Two is a prototype application that, in many aspects, is far from the level of usability found in most commercial applications. However, the two primary shortcomings of Smart Objects One, the abilities to 1) efficiently manage numerous constraints and 2) add and remove constraints on the fly, have been resolved in this second implementation. These improvements are attributed to the use of Cassowary, which acts as the solving engine for Smart Objects Two, but also inspired changes to the way the applet manages object data (through arrays of object classes). The following chapter lists parts of Smart Objects that we feel could use improvement and concludes this thesis.
Chapter Five: Assessment of Smart Objects

This thesis aimed to illustrate how design intentions and parameters could be explored and communicated through object constraints and behaviors. We presented two versions of a constraint-based 3D design environment, Smart Objects. The design scenarios described in chapters two and four demonstrated how design intentions can be described as constraints and manifested as behaviors. This final chapter discusses the benefits and shortcomings of Smart Objects and possible future directions.

5.1 Object Behaviors
In Smart Objects, constraints manifest as object behaviors. In the examples described in both Smart Objects One and Two, a design problem’s absolute requirements (e.g., building codes, spatial needs) are assigned behaviors that change an object’s physical properties (e.g., size and position), while the design problem’s flexible preferences are assigned behaviors that change an object’s display properties (e.g., color and opacity). This ensures that the essential relationships between objects are always maintained, while it allows designers to explore forms outside of their stated preferences. For example, the resizing beams in the first design scenario in Smart Objects One maintains the absolute requirement of structural integrity (Figures 2.5 –2.7). On the other hand, the Vanna Venturi House stairwell becomes translucent to allow a designer explore forms that are not symmetrical, contrary to a design preference. It is important to allow a designer to violate design preferences because preferences within a design problem will often be contradictory (Venturi, 1966). Instead of rendering an inflexible model, Smart Objects alerts the designer when a preference is being met or violated by changing the values of the object’s material.

5.2 Constraint Communication
Some of the examples described have used additional 3D elements in the VRML world to help the user better understand the design intention and constraint that is triggering an object behavior. For example, in the second design scenario (solar obstruction), a clue is needed to help the user understand why a building’s color changes when it is moved (Figures 5.1 and 5.2). In Figures 5.3 and 5.4, the addition of a translucent yellow plane helps the user (perhaps a collaborator who did not specify the constraint) understand that a pink building means it is blocking sun falling onto the park. In the Vanna Venturi example, a red line marks the centerline of the composition. When a user (collaborator) moves a façade wall, the opposite façade wall moves as well (Figures 4.3 and 4.4). The presence of the centerline helps convey the symmetry constraint.
Object behaviors are necessary to maintain relationships and indicate the consequences of a design move. The addition of what could be called “diagrammatic objects” helps to clarify a designer’s intention to a collaborator. Other means to help explain object behaviors could be adding diagrammatic figures to the 2D rendering of the model or simple text warnings.

5.3 Designer/Model Interaction and Constraint Types

Object translation is currently the only available method for object manipulation in Smart Objects. In the future it would be easy to add more methods by which a designer could interact with a model, such as object rotation and resizing.

As the number of ways a designer can interact with objects increases, so should the types of constraints we can apply to objects. For example, a designer might want the two walls of a hallway to remain parallel (Figure 5.5). With a rotation constraint between these two objects, as one wall is rotated, the other rotates the same amount (Figure 5.6). In another instance, the size of a window is proportionally constrained by the wall it is set within (Figure 5.7). As a designer increases the height of the wall, the window also gets taller (Figure 5.8).
We are also interested in constraining different types of object manipulations to one another. Figure 5.9 depicts a wall and building mass. The edges of each object are constrained to line up. Therefore, when the user rotates the building, the wall moves to maintain the edge constraint (Figure 5.10).
5.4 The Functionality of Smart Objects

This section discusses what improvements would need to be made to Smart Objects to make the application easy to use by the architects and their clients.

We chose VRML as the 3D interface for Smart Objects because a designer using commercial modeling applications like FormZ and 3D Studio Max can simply save their models as VRML files and bring them into Smart Objects. However, before we can take, for example, a FormZ generated VRML file into Smart Objects, we must slightly alter the VRML code. Alterations include renaming the nodes of every object and color, and adding TouchSensor and PlaneSensor nodes to every object. The preparation procedure is not difficult, but it is time-consuming. In the future, we could add a translation and preparation function to automatically convert a CAD generated VRML file to a Smart Objects ready file.

Smart Objects is not a modeling program. However, the ability to add and remove objects from a model would be welcome. A Java applet written by Thomas Jung enabled objects to be numerically specified and added to a VRML world (Jung 2001). Ken Camarata and I wrote another applet that builds on Jung’s work; it enables a designer to sketch a 3D object and add it to the model (Figure 5.11) (Eggink 2001). We could add this function to Smart Objects in the future.
Figure 5.11 A Java applet (by Ken Camarata and Dustin Eggink) models a sketched box in a VRML world.

Smart Objects currently does not save any changes made to a model. The objects a designer moves and the constraints added or removed will not be represented the next time it is opened. In the future, Smart Objects should include an option to save altered models and constraints to a new file.

5.5 Conclusion

This thesis sought a way for architects to communicate and maintain their intentions throughout the design process. We recognize the rich design knowledge architects bring to a building project. We also acknowledge the needs and wants that a client contributes to a design problem as well as the absolutes that require a building to be safe and firm.

Architecture is a collaborative process. Architects, consultants, clients and contractors work together to create a building that will meet the needs of its inhabitants and users. An important role of an architect is to communicate how a formal solution resolves the problem at hand. If the problem was simple, this task would be easy. However, architectural design problems are, as Venturi states, “complex and contradictory.” Despite an architect’s best attempts, some of the intricacies of the problem can become lost as the form of a building is resolved.

Smart Objects is a prototype software application. However, we see this approach as a potential aid to a team of architectural collaborators. Architectural forms are related to the design problems they resolve. By representing design in terms of objects, constraints and object behaviors, Smart Objects is a possible method to ensure that these relationships are maintained and understood by all members of a design collaboration.
We see the architect as not just a maker of form, but also the definer of the form’s problem context. The expertise of an architect lies in her ability to, as Alexander states, “achieve fitness between (these) two entities.” Smart Objects seeks to extend this expertise for a designer and her collaborators.
References


