INTEGRATIVE AND INTERACTIVE APPROACH TO THREE-DIMENSIONAL PROGRAMMING

By

HYUNJU SHIM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2006

To my parents and brothers

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Paul A. Fishwick, who gave me inspiration and guidance throughout my Ph.D. studies at the University of Florida. I would also like to give my sincere appreciation to my Ph.D. committee members, Dr. Abdelsalam A. Helal, Dr. Benjamin C. Lok, Dr. Howard Beck, and Dr. Joachim Hammer, for their precious time and advice for my research. Special thanks go to Peter J. Dobbins and Jonathan Ohlrich for helping me to successfully finish the empirical study with their classes. I appreciate all the colleagues in our research group for sharing valuable ideas. Also, I am grateful to the National Science Foundation and the Air Force Research Laboratory for their financial support for my studies. I owe great love to my parents, who encouraged me throughout my studies and my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

INTEGRATIVE AND INTERACTIVE APPROACH
TO THREE-DIMENSIONAL PROGRAMMING

By

Hyunju Shim

August 2006

Chair: Paul A. Fishwick
Major Department: Computer and Information Science and Engineering

This dissertation introduces an integrative and interactive environment for three-dimensional (3D) programs. In this environment, users can lend themselves to the creative activity in designing their own icons for the basic building blocks of programs, and they can also wire them together to construct a program. This work is viewed as providing a user–customized method of programming while leveraging the ability to blend both program syntax and semantics within the same virtual space. The integrative approach has been broadened to simulation modeling, especially for system dynamics modeling. A new method for modeling the particular type of a predator-prey system is demonstrated by leveraging the power of human interaction, as applied to two different models: a system dynamics model and the geometry model. Both models are represented in three dimensions and are fully integrated so that interacting with one model's components provides context for the other model's components through a method of highlighting. With the integrative modeling approach, the linkages between the related objects from different model types are visualized by effective interaction means.

As a way to leverage the integrative and interactive environment for visual programming, a simple visual programming language for constructing 3D graphics, Visual Programming Tinker (VPTinker), was developed. VPTinker is a data-flow visual programming language for novices to construct virtual Tinkertoy models. It is an iconic language using a factory metaphor representation, and its programming is achieved by wiring the icons together. The integrative environment allows the output of the VPTinker program to be integrated with the source program within the same virtual space. This approach to graphics programming enables a programmer to have a holistic view of the program syntax, the source program, the semantics, and the program output. In the VPTinker environment, a program is arranged within the two–dimensional (2D) plane, and the program output is constructed in the 3D space. New interaction mechanisms among the source program, the output model, and the programmer have also been designed. The user interactions include simultaneous highlighting of related components between the source program and the program output; progressive animation for program execution and its output construction; the play–pause–and–stop execution modes; and adjustable speed of the execution and animation. The VPTinker environment also includes a novel debugging feature, such as setting 3D breakpoints among the program components for the step–by–step execution.

Classroom assessments were performed in the Spring and Summer 2006 semesters at the University of Florida to evaluate the effect of visual programming on the problem–solving capability for graphics programming. To evaluate the graphics programming in VPTinker by comparison, the textual counterpart language was designed in such a way that it is mapped one–to–one with VPTinker. The empirical results showed that the subjects who used VPTinker received better grades than those who used the textual counterpart language. The empirical results were analyzed based on the cognitive dimensions of visual and textual notations.

CHAPTER 1
INTRODUCTION

1.1 Motivations and Challenges

Visual programming refers to programming that makes use of visual means in the process of programming. The goal of visual programming is to improve the problem–solving capability in programming by replacing traditional textual notations in programming languages with visual ones. Ever since the introduction of the seminal research in visual programming [1, 2], much research has been conducted regarding the generation of effective visual representations and tools for different programming paradigms and tasks [3–5]. Until the late 1980s, the prevailing techniques among visual programming communities were in two dimensions [6–8]. It is only recently that three–dimensional (3D) visual programming was introduced and gained interest [9–12]. Most research in visual programming focuses on easiness, efficiency, and improvement of programming. The success of visual programming languages (VPLs), as a general programming language, is controversial, and empirical studies on the usability test of VPLs have shown conflicting results [13–17]. Unlike general purpose programming languages, VPLs have been successful as *domain–specific languages* from various areas [18–21]. However, leveraging the 3D graphics in the programming process itself for graphics programming has not been studied widely, and it has remained an interesting question. Specifically, this research was initiated with the following five motivations:

- How can creativity increase peoples interest and help their learning of programming?
- Why not leverage 3D computer graphics in the programming process?

- Can a 3D program be visually integrated with its output 3D graphics?
- How can the user leverage 3D execution and debugging for 3D programs?
- Would the visual approach improve problem–solving in graphics programming?

The first motivation is about customizing visual expressions in visual programming. In designing icons to represent language constructors in visual programming, primitive objects are commonly used as language expressions. The layouts are optimized in such a way to minimize line crossings and maximize symmetry. There has been little freedom for programmers to customize the appearance of visual expressions, and issues such as stimulation of the user's creativity and aesthetic perspectives of visualization are often neglected among visual programming communities [22]. As computational technologies diffuse into our everyday lives and aesthetics are discussed as equally important as usability, the need to consider aesthetics in the design of interactive systems increases [23]. Recently, within the Human–Computer Interaction (HCI) community, an increased role has arisen in the importance of properly balancing utility with aesthetics. Aesthetics are now discussed as a distinguishable, measurable construct in the field of HCI [24, 25]. This dissertation introduces 3D customizable icons in visual programming to stimulate the user's creativity and aesthetic perspectives. This approach to programming has particular utility in education where creativity has been shown to increase subject interest.

Besides the customizable approach to 3D icons, as a second motivation, we were interested in a visual approach to graphics programming. That is, an input program and its execution output are in the form of 3D graphics. The programming code to generate computer graphics artifacts usually involves learning and coding in a language such as C++ or Java. In the creation of computer graphics programs, the underlying application programming interfaces (APIs), such as OpenGL [26, 27] and Java3D [28], provide a set of textual statements.

These statements allow the specification of geometric objects using the provided primitives, together with a set of commands that control how these objects are rendered. For generating complex 3D computer graphics, authoring tools such as Maya, 3D Max, and Blender are used. These approaches are effective, but the question remains why we are not leveraging the power of 3D computer graphics *in the programming process itself*.

Having an input program and its output as 3D graphics, our third motivation was about creating an integrative and interactive environment in which both the input program and the output graphics co–exist within the same space. A source program and the execution are two different perspectives of problem–solving. Each of these perspectives describes a static and dynamic perspective, respectively. While they are semantically tightly coupled entities, little research has been conducted from visual programming communities about surfacing the relationships between the program source and its output in visualization forms. From the HCI community, several techniques have been introduced to assist comprehension of a complex system that has more than one conceptual perspective. Multiple views are a technique that uses more than two distinct views to support the investigation of a single information system [29]. Several techniques, such as linking and brushing, are introduced for displaying multidimensional information using multiple views [30–35]. Multiple views allow viewers to focus on partial information while they are linked so that the information contained in individual views can be integrated into a coherent view of the information as a whole. In the computer simulation community, various model types have been developed and introduced to describe different aspects of these systems. A system can be modeled by views based on its logic, dynamics, or physical appearance. Recently, increasing interest has been shown in integrating different model types of simulation for visualization [36, 37]. The integrative environment allows programmers to have a holistic view on both

the program source and its execution output, and also to experience special types of interaction between the program source, the execution output, and the programmer.

As a fourth motivation, we were interested in creating novel interfaces for program execution and debugging that leverage an integrative approach. When writing a complex program, programmers are likely to make syntactic or semantic mistakes. Debuggers assist programmers to identify and resolve program bugs with such features as single–stepping, n–stepping, trace, breakpoints, and watchpoints. According to Johnson and Kenney [38] and Johnson [39], a debugger should provide more than just information. It should be easy and it should allow users to focus on examining a program itself rather than controlling the debugger. User interaction should be minimized to request information, and the format of reported data should be quickly interpretable. Using debuggers, programmers can easily monitor the status and values of variables and memory, and they can run the program instruction–by–instruction or by a group of instructions.

With textual languages, most debugging interfaces are in two dimensions. Interactions with such debuggers are provided by streams of strings, menu–bars, drop–down menus, and pop–up windows. Information related to debugging is delivered in the textual form usually provided by a separate console or window. Three–dimensional debugging of 3D programs enhances user interaction via leveraging motions and animation, which are fundamental features of 3D programs. Recently, increasing research has been conducted in leveraging 3D animation for illustrating the execution status of programs [10, 12, 40, 41]. With the integrative environment, programmers construct, execute, examine, and debug 3D programs within the environment where the program source and the execution output are presented together. This integrative environment can be further broadened for the

simulation model types in such a way that different model types of the same system coexist within the same space.

Finally, as a fifth motivation, we were concerned about whether or not programming for computer graphics (i.e., programs, when executed, would create geometry and rendered scenes) would be improved by a visual approach. Computer programs are very complex systems, and learning programming for problem–solving is subject to the characteristics of the language in which the solution is described [42, 43]. For the same programming task, a programmer constructs different mental models with different programming languages [44, 45]. In [14], it is shown that the effectiveness of language is task–dependent . VPLs may be easier to learn and understand than textual languages. The evaluation of a visual programming system is crucial, but the usability test of visual programming is not trivial. One way of evaluating visual programming is to compare it with a traditional approach, which is programming via a textual language. We were interested in investigating the problem–solving capability in visual programming for graphics programming by comparing a visual language to its textual counterpart language.

1.2 Contributions to Knowledge

1.2.1 Flexible Program Visualization

Until recently, due to the high cost graphics hardware, visual expressions in visual programming language were adhered to primitive objects. Qualitative perspectives in visualization, such as aesthetics, creativity, and desirability, therefore, have been ignored from the visual programming community. However, with today's low cost and high performance graphics resources, the user's creativity and aesthetic perspectives are becoming new features in visualization. Three–dimensional representations provide a greater number of degrees of freedom not present in two dimensions, or at least extending what is available in two dimensions for aesthetics and expression. The degrees of freedom for three

dimensions include one extra positional dimension; two additional degrees of rotation; an addition of presence and immersion; an increased engagement, which is the ability to interact with a scene; an ability to texture and add rendering effects; and a creativity "index," that is, the ability to extend the creative potential beyond two dimensions.

This research introduces a customized approach to visual programming that enables novices to learn programming through a high degree of creativity by expanding the aesthetic range of program representation. We express this creativity in our program constructors through the use of customized 3D icons. The broader view encapsulates two ideas: form and function have equal footing and we should strive beyond pure optimal functionality toward environments that balance look and feel with pure usability. With this approach, users are exposed not only to the programming environment in which they can wire basic building blocks together for programming, but also can design their own icons, which lend themselves to the creative activity. A framework has been developed for programmers to leverage their own designed icons in programming. In this framework, dynamic model types, such as the Functional Block Model (FBM) and Finite State Machine (FSM), are applied in the construction and execution of programs to capture the nature of programming principles.

1.2.2 Graphics Approach to Graphics Programming

For programming tasks that require spatial reasoning, a visual approach provides close mappings between the problem domain and the program domain, therefore enhancing problem–solving capabilities. A visual approach allows programmers to focus better on the flow of data and spatial reasoning than sequential and textual approaches allow. This dissertation introduces a 3D visual programming language, Visual Programming Tinker (VPTinker), for computer graphics programming that leverages the data–flow paradigm and a

factory metaphor for its conceptual model and representation. The VPTinker language programming applies a visual programming approach to learning and practicing data–flow graphics programs. The use of a factory metaphor in program visualization naturally and intuitively reveals the concepts of a data–flow paradigm and surfaces it in the program configuration. The use of a factory metaphor naturally lends VPTinker programs to the 2.5–dimensional (2.5D) layout where icons are laid on the floor. The limited movement of 2.5 dimensions is less confusing, and it provides intuitive mapping to two–dimensional (2D) interaction devices, such as a mouse. One advantage of the 2.5D layout for VPTinker is that novices can easily grab and move icons using a mouse while freely navigating the 3D program space without special training. Using this language, the construction of a Tinkertoy model is represented as a data flow network. The visual approach to data–flow programming surfaces the flow of data in the program's configuration, which makes it easy to learn and easy to practice data–flow paradigms.

1.2.3 Integrative and Interactive Approach to Programming and Modeling

In this dissertation, we introduced an integrative approach to visualization where interrelated systems are presented within the same virtual space. An integrative environment for VPTinker programs blends both program syntax and semantics within the same virtual space, and it enhances the interaction among the source programs, the program execution outputs, and the programmer. Within this environment, an approach to graphics programming begins with the user constructing a 3D graphics program. Execution of the program generates 3D animated graphics, which are juxtaposed with the input program. The integrative approach allows programmers to navigate the scene that contains both the program source and the output and to view the scene from various viewpoints. With traditional graphics programming approaches, the source program and the output

graphics are two distinctive entities. With the integrative approach, the program source and the output graphics can be treated as coexisting entities. Using the integrative environment, programmers can watch the output graphics being built as the input program is executed and animated. This suggested mode of operation integrates *syntax* with *semantics*, where *semantics* is loosely specified to be equivalent to *program behavior*. As the program execution progresses, semantically related objects from the program source and the output are simultaneously highlighted. This type of visualization provides a holistic view of program execution. With the integrative approach, the power of human interaction is also applied to the program source and the output in such a way that interaction with one model's components provides context for the other model's components through a method of visual highlighting. The dual interaction between both models is a form of integrative modeling where interrelationships among model components are surfaced in the user interface. This dual interaction allows free form exploration of both models and their equivalences. With dynamic scenes that offer users interaction and response, users become more immersed in experiencing a scene. In visualization, interactions help users to comprehend abstract concepts or models. The interactive visualization can also be used to reveal hidden information on users' demand or to stimulate users' interest, which is an important factor in learning.

This dissertation further leverages the integrative and interactive environment for simulation model types. With the integrative approach, different model types coexist and they are linked in the same virtual space. The integrative visualization reduces efforts for context switch between different model types. The explicit component identification for related model elements supports the understanding of the mappings between different model types presented in the visualization.

1.2.4 Three–dimensional Execution and Debugging

An intelligent learning environment has been developed for programming and debugging 3D programs. Within this environment, 3D programs are progressively constructed and interactively examined in 3D space. The status of the partly or completely executed program is demonstrated by visual means or animation. Programmers are allowed to place metaphoric walls among the program constructors to set breakpoints. Animation of program execution allows programmers to visually trace the execution of a program for each breakpoint. Within the integrative environment, the related components from the source program and the execution output are simultaneously highlighted on a programmer's demand. Debugging is enhanced by play–pause–stop execution modes that let programmers stop the execution and resume the execution for inspecting partly constructed execution outputs. Combined with component identification, setting breakpoints and the play–pause–and–stop execution modes allow programmers to check a program fragment easily at any point of program execution. This type of the progressive debugging environment is especially beneficial to novices. A programmer's control over the program execution is amplified via adjustable speed of program execution. Three–dimensional progressive debugging makes it practical to: monitor the program execution intuitively; do early testing and demonstration; and make corrections sooner in the software development cycle.

1.2.5 Evaluation of Visual and Textual Language Notations

One of the main interests of visual programming languages is the evaluation of their problem–solving capabilities. One way to evaluate this is to compare it with the traditional approach, which is the problem–solving capability via a textual language. For the evaluation of VPTinker by comparison, a textual counterpart language is designed in such a way that it is mapped one–to–one

with the iconic language. Designing the textual language starts by specifying the mapping criteria between the visual and textual languages. Since the focus of designing the textual language is for comparison with VPTinker, the designing mappings between them are crucial issues. For an accurate comparison, they must provide the identical capabilities for problem–solving, except for their notations. Everything must therefore be identical except that the syntax for one language is in visual and the syntax for the other language is in textual. For complete mapping between VPTinker and the textual language, a formalism must occur for the mapping transformation process. The mapping criteria between VPTinker and its textual counterpart language are that the mapping should be unambiguous, complete, consistent, and equivalent. Classroom assessments were performed in the Spring and Summer 2006 semesters at the University of Florida to evaluate the effect of VPTinker programming on the problem–solving capability for geometry manipulation problems against its textual version of language. The empirical results showed that the subjects who used VPTinker received better grades than those who used the textual counterpart language. The empirical results are analyzed based on the cognitive dimensions (CDs) of visual and textual notations. Based on the CD analysis, visual notations fit more closely to the mental representations than textual ones for the tasks that require spatial reasoning, and visual programming would better improve problem–solving in graphics programming than textual programming.

1.3 Organization of This Dissertation

Two foci of this dissertation are 1) customization in model and software visualization and 2) the integrative approach to 3D modeling and programming. Previously, a Web– and XML–based framework, the RUBE framework, was developed by former members of our research group. The RUBE framework provides a necessary environment for the construction, visualization, and execution

of simulation models. We expanded the RUBE framework as a programming environment. The RUBE programming environment allows users to lend themselves to the creative activity in designing their own icons to represent the basic building blocks of programs. Users can also construct a program as a data-flow model or as a state-transition model using their own designed icons. Customization in visualization allows users to leverage metaphors and to expand the aesthetic range, which can stimulate their interest and productivity in programming.

The second focus of this dissertation is on the integrative and interactive approach to 3D modeling and programming. The main concern with the integrative approach is to blend multiple visualizations, which describe different perspectives of the same system, within the same space. The integrative approach provides a holistic view and intuitive interactions between the multiple visualizations. We applied the integrative approach to system dynamics modeling, which is a special type of simulation modeling, and to the 3D programming for graphics applications. In modeling, different perspectives of the same system are different model types, such as the dynamic and geometry models. In programming, different perspectives are the source program and its execution. Within the integrative environment, semantically related components from multiple visualizations are fully connected. The relationships between the components are also surfaced by dual interactions via the user interface.

The rest of this dissertation explains details about our approach to visual programming. Chapter 2 discusses related research. Chapter 3 introduces the customizable approach to software visualization. Chapter 4 presents the integrative and interactive approach to simulation model types, especially for a system dynamics model. Chapter 5 provides an integrative and interactive approach to graphics programming using an iconic language, VPTinker. Chapter 6 offers designing of a textual counterpart language of VPTinker and an empirical study

comparing the problem–solving capability of VPTinker with its textual counterpart language. Chapter 7 concludes this dissertation with possible future research.

CHAPTER 2
RELATED WORK

This chapter introduces related works from the area of visual programming and software visualization using metaphors.

2.1 Visual Programming

ConMan [6] is a high–level visual programming language that lets users dynamically build and modify graphics applications. By interactively connecting simple components in ConMan, users construct a complete graphics application. A data–flow metaphor is used in ConMan, and users construct and modify applications by creating components that are interconnected via input and output ports. Components that have been developed for the ConMan data–flow environment include the sweep component to create a surface by sweeping; the tape component for recoding; the render component that supports different rendering qualities from wire–frame to ray tracing; the mixer component to interpolate shapes and color; and the watch component for the inspection of data flowing across the screen. Developers are encouraged to break monolithic applications into functional components that communicate with each other using high level data structures. The current application field of this research is the same as ConMan, which is graphics programming. However, while ConMan is a 2D visual programming environment that takes a data–flow metaphor, this research provides a 3D programming environment deploying different principles from dynamic model types and customizable icons.

Pictorial Janus (PJ) [7, 8] is a 2D visual programming language for parallel logical programming. Programs in PJ are drawings in which the execution is defined as the animation of the drawings. The basic elements of a PJ program

are closed contours and directed and undirected connections between them. Geometrical representations of PJ objects, such as shapes, size, and color, have no semantics. Objects of PJ are constants, list elements, links, functions, agents, and rules. PJ programs are defined by combining PJ objects. Agents in the PJ program communicate with each other via message passing. The behavior of the agent is determined by the preconditions of the rules in case they match the corresponding input patterns of the agent. Janus In Motion (JIM) provides interactive editing, animating, and debugging environment for PJ programs.

Lingua Graphica [9] is a visualization of procedural textual languages. It defines a visual 3D syntax for C++ programs that allow users to inspect and modify the virtual reality simulation code without having to leave the virtual environment. Graphical primitives used in Lingua Graphica are color, translucence, shape, size, associative links, co–location, text, sound, and motion. Associative links are used for representing class inheritance, data–flow binding, and function calling or definition sequence. Co–location refers to the concepts of location, containment, and intersection of one object with respect to another object.

Solid Agents in Motion (SAM) [10] is a visual 3D programming language, visualization, and environment for parallel system. A SAM program is mainly given as a set of 3D objects with no separate textual description. Three–dimensional objects in SAM are 3D messages, agents with ports, and rules with a precondition and a sequence of actions. Input and output ports are distinguished by the direction of the representing cone. In SAM, the order by which rules are elaborated is not defined. The execution of a SAM program is based on a synchronous communication. The SAM execution cycle has two phases: agent execution and communication. The execution of a SAM program can be observed by animation. The SAM program allows users to build the abstract visual representations and the corresponding concrete graphical representations for programs. In the concrete

representations, additional static 3D objects that are not part of the program can be included as a background to give a more realistic environment.

3D–Visualan [11] is a 3D rewritable, rule–based programming language in which both programs and data are expressed by 3D–bitmaps. Programs of 3D–Visualan are the ordered set of pattern–replacing rules. The priority of rules is determined by their locations in the 3D–bitmaps, such as the rear rules which have higher priority than the frontal rules. The behavior of the program is described by means of before–after rules, which define how 3D–bitmaps change over time.

VisuaLinda [12] is a framework and a system which makes use of 3D visualization to help the programmer debug parallel programs, and it also displays both the relation between the Linda server and client and their execution. In this framework, processes are laid out in an XY–plane and a Z–axis represents time. The 3D layout of processes and time lets both the process–relation diagram and the process–time diagram be visualized in the same 3D space. This integrated visualization supports a simultaneous understanding of a process relation and a time flow. Within the 3D space, it is also possible to display a large number of processes and communication links between them without crossing in space.

Programming virtual object behavior in virtual reality Program (PiP) [46] is a virtual environment system in which a user can create, modify, test, and save object behaviors. The PiP program extends the visual programming approach to 3D multimodel programming to represent the behavior of a program or object by manipulating it through the 3D space. A PiP virtual object model consists of its forms, functions, and behaviors. The most typical behaviors of objects are specified by demonstration. For example, to specify a condition that an object must be in a certain position and orientation, the user simply manipulates and moves the object into the desired configuration. Using PiP, users can program or specify the motion

or animation intuitively through a better understanding of direct programming in 3D space.

In Computer Animation Environment Language (CAEL) [47], visual syntax–directed programming within a 3D computer animation context can take place. Language grammar in CAEL augments a subset of Pascal with procedures. Program representation subclasses utilize 2D depictions which are positioned in 3D space. In CAEL–3D, users are allowed to alter the appearance of the geometry and/or color of the polygonal 3D objects, which brings a level of user–specifiability into the representation. A user is free to create 2D and 3D graphical polygonal elements for representing program constructs. The incorporated menu–based interaction methodology and tools of the 3D computer animation system let users act upon and maneuver through the programming constructs.

3D–PP [41, 48] applies the direct manipulation of operations to the 3D program elements. 3D–PP is based on the concurrent logic programming language Guarded Horn Clauses (GHC). The visual program of 3D–PP is composed of a combination of hierarchical nesting boxes of pictorial programming elements, such as atom, list, id_data, goal, and built–in goal. An extended drag–and–drop technique is used for describing the program structure of nesting boxes. The semi–transparent representation, using nesting level filtering and double–click browsing, improves the spatial problem in visual programming caused by a small screen. Debugging of a 3D–PP program is supported by animated execution. Animated execution animates state transitions while execution of the program proceeds. The animation is displayed in a similar manner with the programs, thus the programmer can easily read the animation. Three kinds of functions are provided in combination with animated execution: 1) rewriting for speeding up testing and modification of processes; 2) marking data for tracing data; and 3) marking rules for narrowing suspicious implementation.

A Visual Programming Environment for Python (ViPEr) [49] is a Python– and Tkinter–based visual programming environment for composing networks of Python scripts for molecular data processing and visualization. It allows non–programmers to interactively build a network of computational streams and visualize their data without writing code. In ViPEr, nodes are lightweight wrappers of functionality available in Python and organized in libraries. Drag–and–drop adds nodes into the graphical network editor. Input and output ports of nodes are connected using the mouse to define a directed graph. This directed graph is used to propagate data and execute nodes. Subnetworks can be encapsulated into macro nodes, allowing nesting of networks. Tooltips and balloons provide runtime information about node functions, inputs, and outputs. Data flowing through nodes can be interactively monitored and introspected. Users can specify the appearance of the port icon using different colors or shapes, which provide visual hints for connecting the proper outputs with the proper inputs.

2.2 Metaphorical Software Visualization

The CyberNet [50, 51] project uses 3D metaphoric worlds to visualize the network data. It was developed with adaptive navigation features that are dependent on the 3D metaphor used for visualizing the information and on the user's task. The network data are depicted metaphorically using the visual parameters of the virtual world. Different metaphors are used to visualize the information according to the type of the user's task. The choice of a metaphor depends on the type of information that is relevant for that task; the best mapping between the data values; and the metaphor's visual parameters and user preferences. Metaphors such as a building, a city, a solar system, a cone tree, or a landscape have been designed. The task–dependent navigation, called metaphor–aware navigation, was approached. The metaphor–aware navigation

allows users to navigate in the virtual world with the mechanism most suited to that particular metaphor.

The Information Cube [52] is a 3D visualization technique of hierarchical information that uses 3D cubes. With the cube metaphor, information is represented by a container or a box. Nested cubes represents hierarchical information in such a way that the outermost cube represents the top level data and the next level data are enclosed inside of the outermost cube. Each cube is rendered in semi–transparent color so users can easily see the inside of the cubes. The semi–transparent rendering also lets inner level boxes fogged out gradually and complex hierarchical information systems maintained reasonably.

Rilling and Mudur [53] introduced the use of metaballs to provide 3D visual representations of software systems. Metaballs are a 3D object modeling technique which blends and transforms an assembly of particles with associated shapes into more complex 3D shapes. They are also called metablobs, soft objects, and point clouds, and they are extensively used in representing complex organic shapes and structural relationships in biology and chemistry. Rilling and Mudur applied the metaball metaphor to visualize software entities and their dynamics. Particles in a metaball metaphor are mapped into software structures, with blobs representing an object or a function. The potential energy around blobs is mapped into the strength of the coupling between entities. Dynamic aspects of software are visually depicted by animating the appearance properties of blobs. By mapping software entities or parameters in the software slices with metaball models, Rilling and Mudu created a 3D virtual space where users can walk around for examination of the model.

Panas et al. [54] introduced a 3D city metaphor in visualizing complex software for the purpose of maintenance and reverse engineering. They mapped static and dynamic properties of software to the city environment. The size of the buildings

represents the amount of lines of code, and the density of buildings shows the amount of coupling between components. Structures and qualities of buildings represent the quality of the system's implementations. Cars moving through a city indicate a program run. Dense traffic indicates heavy communication between various components. Speed and the type of cars represent performance and priority. Car crushing shows program exceptions. In addition to these visualization for reverse engineering, they applied a cost–focused metaphor over the 3D city metaphor to visualize additional business–related information for different types of stakeholders, such as developers, designers, vendors, and project managers. For example, the components currently being modified are colored in yellow. Parts of the system that are no longer needed are colored in brown. Frequently used components are surrounded by fire. Frequently modified components are visualized as a building with flashes.

The Zeugma system [55–57] is an environment for constructing and experiencing Analogical Representation of Programs (ARP). Zeugma allows users to visualize Lisp programs using analogy or metaphor, such as a solar system, cities, and spiders. Given a structured schemata describing relations between program analysis and representation features, Zeugma automatically constructs analogical representation for the given programs. With the Zeugma system, it is possible to switch from one representation to the other for programs. The combined and simultaneous views of both the visualization and the control–flow view provide links between program components and and their visualizations.

Knight and Munro [58] introduced the use of a world metaphor as a tangible object for software visualization within a virtual reality environment. They introduced the world metaphor in which the entire software is mapped into the world, source files are mapped into cities, classes are mapped as districts, and methods are mapped into buildings. Colors of the buildings are used to represent

access controls, such as public, protected, and private. The height of a building represents the lines of code. Parameters of a method are represented by doors. To assist the navigation and orientation of the user's visualization system, a central garden and enclosing fence are included in the visualization, and block structures are used for layout.

Hopkins and Fishwick [59] introduced an architectural and an agent–based metaphor in the representation of an MINIX–like OS with four layers of user processes, server processes, input/outout (I/O) tasks, and process management. In this representation, the operating system (OS) is represented by a building, and each layer of the OS is mapped into one floor of the building. Synthetic human agents are used to represent each task. Message passing between the OS components is represented by a transporting courier agent with a briefcase. The OS devices are mapped into an agent standing behind a desk. Waiting lines of agents are used to represent a device queue. To represent the dynamics of OS which can be modeled by Finite State Machine (FSM), a dynamic traffic light metaphor is used.

Software Landscapes [60] visualizes the structure of large software systems using the landscape metaphor. Three–dimensional landscape elements, customized layouts, and hierarchical network structures are used to represent program entities, their hierarchy, and relationships, respectively. The landscape metaphor, with 3D objects on a 2D layout, facilitates navigation and comprehension of visualizations. The hierarchy of software packages is represented by nested spheres. Spheres contained within another sphere are arranged on a 2D circle. Transparency of spheres are dynamically managed by the distance of the viewer. Spheres are opaque when the viewer is very far and completely transparent when the viewer is very close. Classes are represented by circular discs. Methods and attributes of a class are represented by cuboids on the disc. For clarity, the 3D routing lines are used

to represent the relations between software entities according to their hierarchical levels.

2.3 Differences between Related and Our Work

We can differentiate our work from related research as follows:

- Customizable approach to visual programming: Traditionally, expressions in visual languages have been limited to primitive objects. Recently, several research including [55–57] and CyberNet [50, 51], introduced the use of variety themes and metaphors in software visualization. In these systems, however, a user's freedom in representation is still limited to the pre-existing themes and metaphors. We introduced a customized approach to visual programming that enables programmers to design their own icons to represent the basic building blocks of their programs.

- Visual approach to graphics application: Conman [6] introduced a 2D data–flow approach to 3D graphics construction. We introduced a 3D data-flow approach to 3D graphics construction.

- Integrative visualization: In this work, we introduced an integrative approach where the program source and its execution output co–exist and fully connected within the same space. This visualization approach has not been introduced to the visual programming community.

- Three–dimensional debugging of 3D programs: Unlike debugging textual programs, debugging 3D programs has not been studied widely. Debugging 3D programs, however, is getting more of interest. VisuaLinda [12] introduced 3D debugging by visualizing erroneous components during the program execution. Our work introduced play–pause–and–stop execution modes and setting breakpoints for 3D programs that allow programmers to check a program fragment easily at any point of program execution.

- Progressive animation of program execution: Increasing research in visual programming has leveraged animation to demonstrate the execution status of programs [7, 8, 10, 41, 48]. Besides demonstrating the execution status, we leveraged animation in such a way that the program output, which is 3D graphics, is being constructed as the input program is being executed and animated.

CHAPTER 3
CUSTOMIZABLE VISUAL PROGRAMMING VIA DYNAMIC MODELING

In this chapter, a visual programming environment, which facilitates a customized 3D application programming interfaces (APIs) using dynamic model types, is presented. The environment introduced in this chapter is developed for users to lend themselves to the creative activity in designing their own icons for the basic building blocks of programs and to wire them together to construct a program. The environment also integrates the output graphics to be placed with the 3D input program. RUBE, a Web– and XML–based modeling and simulation framework, provides a necessary environment for the construction, visualization, and execution of program models. In the RUBE framework, users can construct their program models, which will be stored in XML format and translated into an executable program by the translation engines. The RUBE framework makes use of a 3D tool, Blender [61], to provide a modeling environment. Users can use pre–existing icons or create their own to represent program modules.

The unification of 3D dynamic model types with the program construction brings several benefits in programming:

- Capturing dynamics of programs using simulation model types
- Use of metaphoric icons, which increases understandability and productivity of novice programmers
- Use of metaphors to allow greater flexibility and freedom in model representation
- Stimulating user's creativity in the design of program icons
- Leveraging aesthetics aspects

- Juxtaposing program models with program outputs in the same space

3.1 Flexible Approach to Visualization

Designing notations of language constructors is one of the main issues in visual programming. For software and information visualization systems, it is common to use metaphors for the representation of abstract concepts [50–60]. This section introduces a new approach to visual programming in which users are allowed to create their own visual notations for language constructors.

3.1.1 Metaphors in Visualization

The word "metaphor" is derived from the Greek word "transfer." The metaphor's primary function is understanding new concepts from familiar ones. The main functionality of metaphor is in its usability. Lakoff and Johnson [62] demonstrated the pervasiveness of a metaphor in all aspects of human activities not only as a matter of a language but also as a principal way of reasoning and learning. A metaphor can benefit both a novice and an expert in some system. A metaphor provides insights to novice users regarding the nature of a function or application. A metaphor turns to a dead metaphor when it has occurred so often and become a new meaning of the expression. Dead metaphors have a communicative value for experts [63]. Visual programming can take advantage of metaphoric icons, especially in their visual representations. Icons were first introduced in programming when Smith constructed a special user interface for operations in programming language, PYGMALION. He used icons to subsume the notions of variables, reference, data structure, and functions [2]. Inspired by Smith's work, Xerox introduced the 8010 "Star" Information System with the desktop metaphor icons as a user interface [64]. Several research leveraging metaphors in visualization were introduced in Chapter 2.

The conjunction of metaphors in visual programming could generate a synergy effect on learning programming. The use of metaphoric icons, from

which their functionality can be visually referred to, would improve productivity and understandability of programming for novice programmers. Meyer and Masterson [65] claimed that the use of a factory metaphor for the data–flow programming is simpler than flowcharts. Properly coupled with object oriented concepts, a wider group of programmers–those whose minds more easily grasp the factory metaphor–will appreciate visual programming. Some arguments exist that metaphors should not be treated as a necessary criterion in designing visual representations. Blackwell and Green [66] conducted an experiment about the usability of metaphors in visual languages and claimed that metaphors do not particularly assist visual languages.

Examples of metaphoric icons, which might be suitable to be used in visual programming, include a warehouse metaphor for database applications where data are stored and retrieved frequently; a plumber metaphor or a factory metaphor for data–flow programming where data flow from one functional unit to another; and a chemistry metaphor for state–based applications where the current state of a system is determined based on the external or internal conditions.

3.1.2 Aesthetics in Visualization

In addition to usability, aesthetics is another dimension in visualization. Aesthetics in visual programming has traditionally tended toward the manifestation of optimality criteria, such as minimal line crossings and the efficiency of layout. However, several recent publications included in [67] suggest that aesthetics for products, including the visual programming products, should be defined to more broadly cover both form and function or visual appearance and pure usability. This more balanced approach is common in product design [68, 69], and since programs are human products, such as houses and automobiles, we advocate extending aesthetics in their visualization.

Fogarty et al. [23] claimed, as computational technologies move beyond the confines of the work environment and into the rest of our lives, that the need to consider aesthetics, or, more generally, desirability in the design of interactive systems, will increase. Recently, within the HCI community, an increased role has arisen in the importance of properly balancing utility with aesthetics. A successful product is one that achieves both function as well as form. The existence of aesthetics is discussed as a distinguishable, measurable construct in the field of HCI, and it is advocated to better balance major design dimensions regarding usability and aesthetic considerations [24, 25]. The use of personalized and customized metaphors in 3D visual programming and software visualization can stimulate users' interest and creativity. This lets users tailor the appearance of program components while preserving their functionality. The design of personalized and customized 3D icons is one layer where aesthetic aspects can be integrated within software visualization. Three–dimensional representations provide a greater number of degrees of freedom for aesthetics and expression [70]. Customization and personalization in HCI might be one solution to remove some limitations in aesthetic designs caused by their dependency on personal experiences, language, and culture. Another layer of aesthetics in software visualization is applying different textures and rendering methods to the model constructors and the model scene. Using metaphoric icons and adding graphical effects, such as transparency and shading, we can build 3D dynamic systems or program models that might be aesthetically expressed and appreciated.

Figure 3–1 shows the use of customized icons in a 3D program model developed using the RUBE framework. Icons shown in this figure, from the left–hand side, are Texture, Duplicate, Transform, and Display. Different materials, textures, and rendering methods are applied in the creation of this figure. The left figure in Figure 3–1 is a scene from a 3D program with colored materials and

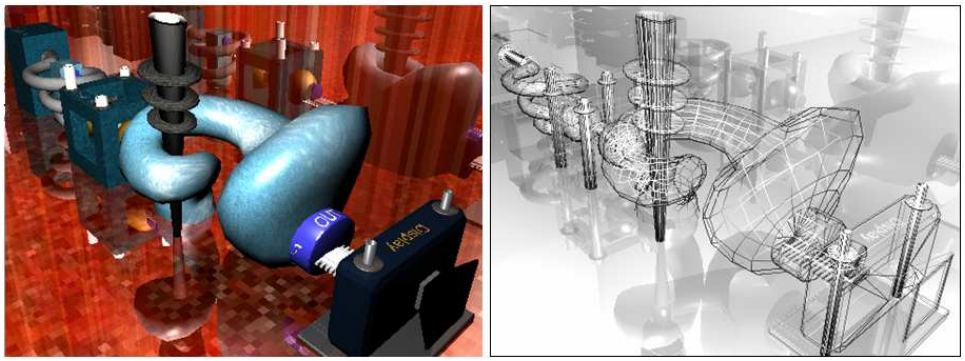

Figure 3–1. Aesthetical rendering of program icons

different textures. The right figure shows the same program and metaphor with no textures but with transparent materials and the wire frame rendering provided by Blender.

3.2 Modeling Programs via Dynamic Models

A framework, called RUBE, has been developed for programmers to leverage their own designed icons in programming. In this framework, dynamic model types such as the Functional Block Model (FBM) and Finite State Machine (FSM) are applied in the construction and execution of programs to capture the nature of programming principles. Before introducing the RUBE framework, we will examine the principles of modeling and programming as a way to leverage modeling techniques in programming.

3.2.1 Programming versus Modeling

Programming is the process of creating a computer program with instructions that a computer can interpret to solve human problems. In general, programming is classified into two categories, declarative programming and imperative programming, depending on the model of computation. Programming languages are further categorized into paradigms that include functional, data–flow, logic, Von Neumann, and object–oriented [71]. Each paradigm reflects different perspectives of a model

of computations from which a programmer can conduct problem–solving in more logical and efficient ways.

Modeling in general refers to the creation of the representation for a certain system which involves reasoning about the system. The goal of modeling is to come up with a representation of a system that is easy to use for describing the system in a mathematically consistent manner [72]. Following efforts to discover concepts and representations of models, the first general purpose simulation languages, General Purpose Simulation System (GPSS) [73] and SIMSCRIPT [74], were introduced in the early 1960s. Ever since the early 1960s, concepts of modeling and simulation have been reviewed to promote a more coherent representation and to capture more logical and precise dynamic behaviors of a system. As the principles of modeling and simulation are refined, the focus of simulation model development has shifted from programming to modeling [75].

Although computer programming and modeling have gone in two different directions, they share a common interest: problem–solving. However, while problems in computer programming refer to general problems, those in computer simulation are usually related to an activity about making decisions. For this reason, we can say that modeling is a particular way of problem–solving, and the field of modeling and simulation has developed different computational models from those of programming [76].

While the field of modeling and simulation departed from general purpose programming as it matured, much research has also been done relating programming with modeling to improve or supplement programming. SIMULA is the first object–oriented language and a good example of how modeling and simulation principles can improve programming. The concepts of class, subclass, and inheritance in the object–oriented paradigm were first introduced in SIMULA. Bloss [77] claimed that traditional imperative languages are poorly suited for

modeling the concurrent logic in simulation, and the absence of an explicit time–flow mechanism in the functional programming would fit nicely into the simulation modeling where time flow is an implicit part. Dynamic model types, such as Petri Net and FBM with synchronous and asynchronous properties, are more suitable to represent and practice logical and/or concurrent programs than traditional imperative languages. Recently, enterprise level software is getting complex from one system to multi system, from one programming language to multi programming language, and from isolated network into interoperable networks. The Object Management Group (OMG) announced Model Driven Architecture (MDA) as a solution for effective development of complex software. As its name implies, MDA is a model–centric development approach, which encourages efficient use of system models in the software development process.

Models are abstractions of physical systems which allow engineers to reason about the system under study. They can focus on relevant factors while ignoring unnecessary details. Modeling software enhances modularity of software components while decoupling unrelated components. Models from a classical systems theory include sets of inputs, outputs, and states, along with the transition functions. Defining software with inputs and outputs, along with the transition functions, sets the boundary of software and reduces unexpected errors. Modeling computer programs also open a possibility for computer simulation to be integrated into the process of software development, especially for execution and debugging. The model–based debugging can result in interactive execution and getting early feedback. With this approach, we can also execute and analyze different scenarios against our model. Based on the analysis of the simulation results, such as how different external factors and design decisions affect the system's outputs, we can select the best external factors and designs that meet our objectives. Visual demonstration or animation of the program execution is also easily accomplished.

Figure 3–2. Data–flow between blocks

Animation is a powerful tool to visualize the internal process of a dynamic system, and it has a growing role in computer–aided algorithm design, documentation and debugging, education, performance monitoring, and engineering control.

Unlike most visual programming languages that visually mimic the behavior of textual programs, we applied dynamic model types, such as the Functional Block Model (FBM) and Finite State Machine (FSM), in the construction and execution of programs to capture the nature of programming principles. Unifying modeling with programming brings several benefits to programming: capturing dynamics of programs using simulation model types; getting early feedback by running and modifying models; applying metaphors to allow greater flexibility and freedom in model representation; and storing model components in an ontological model structure [78].

3.2.2 Dynamic Modeling of Programming Principles

The previous section discussed modeling programs at a higher level. This section focuses on modeling of principles and elements of programming and discusses issues in representing programming constructors using dynamic models.

Representing data flow. The concept of data flow is explained as data move around an information system or processor. Visualizing data flow elucidated its concept clearly than textual explanation. In Figure 3–2, arrows represent data flowing into and flowing out of the process unit.

Representing control flow. In imperative textual languages, control flow is often designed to be sequential so that each statement is executed in the order in which it appears statically in the written program. In this case, the sequential composition of statements is the normal style of programming. The

sequential composition, however, becomes unclear when it comes to 3D space. In 3D space, the sequence of statements must be specified for all three dimensions, which is not a simple task. Since the control flow in the declarative programming, especially in the logic programming, is implicit, the representation of the control flow in declarative visual languages is not a big issue. However, for 3D languages wherein the control flow must be specified explicitly, there must be explicit ways, such as using lines or arrows to represent the flow of control between program units. Dynamic model types in simulation have syntactically and semantically well–defined connections between operational units. Use of dynamic model types in 3D programming has advantages with regard to these well–defined connectors, which represent the control flow of programs.

Handling branches and loops. In a structured program, the flow of execution is changed by the evaluation of the conditional expressions in branches and loops, such as "if," "switch," "for," and "while" statements. The conditional branches and loops in programming can be modeled in more than one way using dynamic model types [79]. Semantic similarities exist between conditional branches in programming and FSM in dynamic models. The Finite State Machine (FSM) takes input and decides the next state to move by comparing the input values against the condition of transitions that are going out of the current state. The next state selected by the true transition becomes the current state and the output of FSM. In the RUBE framework, conditional branches can be modeled using a multimodel block that has FSM inside. The condition on each transition in the FSM can model the conditional expression in the branch statements. The input values of the FSM are used to decide which transitions must be taken. This process is identical to the evaluation of the conditional expression of if, switch, for and while statements based on the evaluation variables. Another way of modeling is having FBM with a block that can produce different results for different output

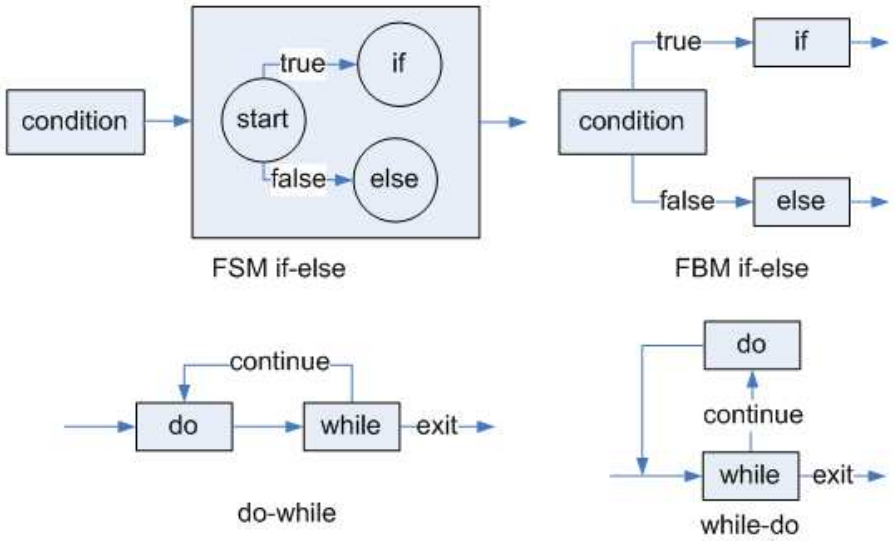

Figure 3–3. Dynamic model components for conditional branches

ports. Loops can be modeled with FBM that have a circular connection between blocks. To prevent infinite execution of the circularly connected blocks in a program model, the exit condition must be specified in at least one block among the circularly connected blocks. The exit condition of loops can be modeled with output–selective blocks or multimodel blocks with FSM inside. The diagrams in Figure 3–3 show dynamic models for "if–else", "do–while", and "while–do" statements, respectively.

Modularization and parameter passing. Program modularization makes large programs more manageable. Fix et al. [80] claimed that the skill recognizing the links tying the separate program modules together can contribute to the formation of the mental representation of a program. With visual programming languages, it is very convenient to represent the program modules and the relationship among them provided by visual scoping and explicit means of parameter passing. In our research, predefined and user–defined functions are stored in separate files. A program is constructed by combining separately defined program modules, which are represented by 3D icons, into a single program

model using the 3D modeling environment called the blender interface in RUBE. Parameter passing is explicitly defined by connecting input ports to output ports between 3D icons.

Level of detail. Usually language constructors in 3D programming require more physical space than textual constructors. In visual programming, the level of details of program modules determines how dense or coarse the program appears in 2D or 3D space. Determining the most effective–but still best representative level of details for program modules–is a very tricky problem. This is because it depends on the complexity of the applications' domains and the skill of the programmers. For visual programming where physical space is a critical factor, different technologies, such as nesting, zooming in, and fish–eye views, provide efficient ways to utilize program space.

Visual execution. One of the advantages in visual programming is the visual execution of programs. Since a program is visually represented, it is easy to animate its execution during runtime. The animation of a program execution can be done by adding a code that changes the appearance of visual program constructors into each of the program modules. In this way, when the specific program module is executed, the appearance of representative icons is changed. When applied to a graphics program, the execution of a program is animated while the result of the program execution, which is the construction of graphics objects, appears. Using the blender interface, a 3D program and its output graphics are co–existed and animated together in the same 3D space.

Concurrency. A program is said to be "concurrent" if more than one execution threads of control exists in the program. For some applications, such as servers and graphical applications, concurrency is a logical structure. Concurrency also arises to handle independent physical devices or to increase system performance. Visualization and animation are also effective methods for

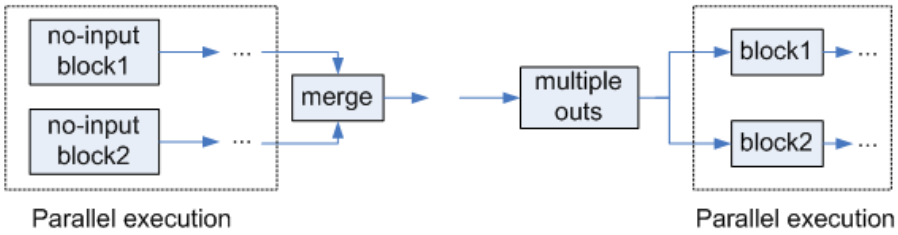

Figure 3–4. Parallelism in functional block model (FBM)

understanding and debugging parallel programs [7, 8, 81]. A line connecting two points is suitable to indicate the transmission and receipt of a message. Zooming or scrolling aids users to comprehend a potentially cluttered display. Animation is good for observing the instantaneous state of the parallel system. The 3D visualization of VisuaLinda helps the programmer in debugging parallel programs and in displaying both the relation between the Linda server and multiple clients and their execution [12]. The use of dynamic model types, such as FBM and Petri–Net, in programming can also facilitate the visualization of concurrent threads and their execution. Figure 3–4 shows concurrent execution of blocks in FBM.

3.3 RUBE Framework

This section introduces the RUBE framework as a visual programming environment, which is developed with approaches of customizable visualizations and programming via dynamic model types.

3.3.1 Framework Structure

Originally, RUBE was developed as a Web– and XML–based modeling and simulation framework for geometry and dynamic models. The RUBE framework includes a Python–based user interface called the blender interface to provide the complete 3D modeling and simulation environment for users [82]. We expanded the RUBE framework as a programming environment by supplying 3D APIs

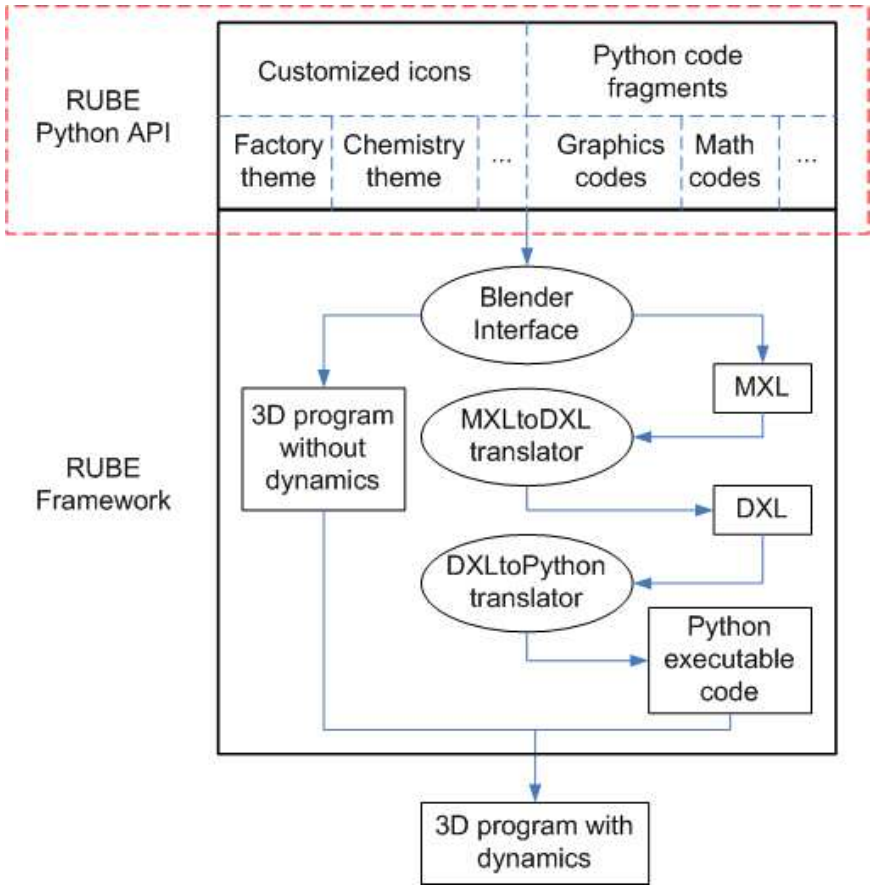

Figure 3–5. The RUBE programming APIs and the RUBE framework

as modeling elements. The 3D APIs in the RUBE framework are composed of program code fragments and their representative icons. Figure 3–5 shows the RUBE APIs on top of the RUBE framework. The detailed translation processes in the RUBE framework are also depicted in this figure. The red dashed–line rectangle in Figure 3–5 shows my contributions for the programming using the RUBE framework.

A user can start programming from the RUBE framework by running the blender interface written in Python from Blender. Once the blender interface is started, users can import the predefined or user–created program modules to construct their program models. This is done by clicking appropriate buttons that represent program modules from the blender interface. When a user locates

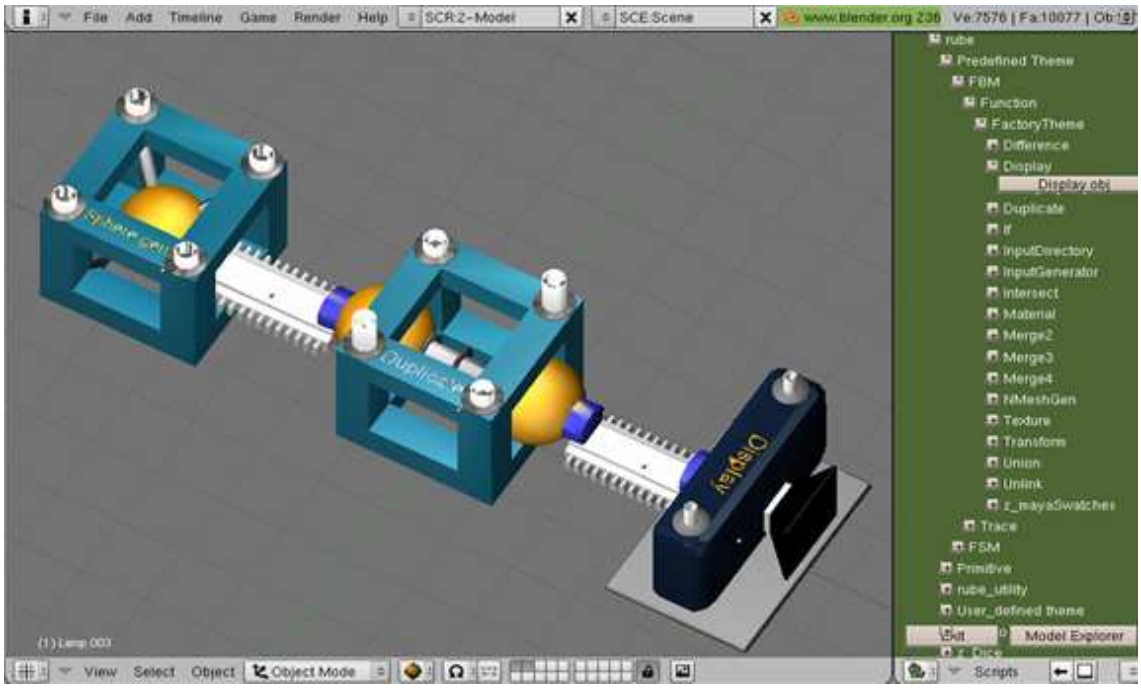

Figure 3–6. Snapshot of the blender interface showing a simple program model

and imports a program module, a representative icon and a piece of Python code associated with the module are also imported into the system. Figure 3–6 shows the snapshot of the blender interface with a very simple program model, which creates, duplicates, and displays a sphere object using factory metaphor icons. The code fragments for the user created program model are scattered throughout the code library, and they are not glued yet to form an executable program.

Internal representations of models in RUBE are in eXtensible Markup Language (XML). By clicking the Simulate button from the blender interface, the model components and structure defined in the Blender 3D window are stored in a textual XML file. The user–defined model is stored in Multimodel eXchange Language (MXL), an XML modeling language developed by the modeling and simulation research group at the University of Florida. Figure 3–7 shows the MXL representation of the program model defined in Figure 3–6. In MXL, predefined or

```
<?xml version="1.0" encoding="utf-8"?>
<MXL>
    <fbm id ="fbm">
     <block id="Generator">
          <output id="Generator_output " datatype="Object" index="0"/>
          <script lang="Python" src="Generator.py" func="Generator"/>
     </block>
     < block id="Duplicate">
          <input id="Duplicate_input " datatype="Integer" index="0"/>
          <output id=" Duplicate_output " datatype="Integer" index="0"/>
          <script lang="Python" src="Duplicate.py" func="Duplicate"/>
     </block>
     < block id="Display">
          <input id="Display_input " datatype="Object" index="0"/>
          <script lang="Python" src="Display.py" func="Display"/>
     </block>
     <trace from = "Generator_output" to = "Duplicate_input"/>
     <trace from = "Duplicate_output" to = "Display_input"/>
      </fbm>
      <simulation start_time = "0" end_time = "10" cycle_time = "0.1"/>
</MXL>
```

Figure 3–7. MXL representation of the program model in Figure 3–6

user–defined blocks in the model are defined as **block** elements. The **trace** element shows the connection between blocks/functions.

The MXL model representation contains information about the model topology or structure. However, the functionalities of model elements are not described. The script attribute of a block element tells where the associated Python code, which describes the functionality of the block element, is located. For user–created functions to be seamlessly glued together into the final simulation code, input and output parameters of the user–defined functions must follow naming rules in the RUBE framework. When the translation engine integrates the pieces of functional codes, it connects outputs and inputs of blocks, as they are configured in the MXL file. The outputs of the source block are then transmitted to the inputs of the proper destination blocks. When creating the functional code for certain icons, users can assume that necessary inputs are provided by an array named input, and outputs from the user–defined functions will be returned properly by storing them to an array named output. The information regarding

which blocks the inputs are coming from and to which blocks the outputs are going is specified in the trace elements of the MXL file.

The values of index attributes in input and output elements of each MXL block are the indices that users can refer to proper inputs and outputs in the correspondent user–defined functions. Multiple inputs to user–defined functions are provided in the input array in exactly the same order as the input elements of the correspondent block element are indexed. The outputs from user–defined functions must be stored in the array named output. Multiple outputs must be indexed and stored in the output array in exactly the same order as output elements of the correspondent block element are indexed.

Figure 3–8 shows how a block, Transform, is defined in MXL with two inputs and a single output along with the associated Python script. This Transform block performs three functions. First, it takes a 3D object and a 3*3 matrix as inputs. Second, it transforms–moves, scales, and rotates–the 3D object based on the values in the matrix. Third, it returns the transformed 3D object. In Figure 3–8, there are two input elements each for a 3D object and a 3*3 matrix. They are indexed as **0** and **1** by the **index** attributes. A single output element, which returns the transformed 3D object, is indexed as **0**. Inside of the **Transform(input, output)** function, which is located at the bottom of Figure 3–8, **input**[**0**] gets its value from the block where the correspondent input element with an index value of **0** is connected. In the same way, **input**[**1**] gets its value from the block where the input element with index value of 1 is connected. In this example, the Python script assumes that a Blender object is provided in **input**[**0**] and a 3*3 numeric matrix is provided in **input**[**1**]. After the transformation of the input object with the values specified in the 3*3 matrix, the object is returned to the next block by storing it in **output**[**0**].

```
<block id="Transform">
     <input id="Transform_input" datatype="Object" index="0"/>

     <input id="Transform_input1" datatype="List" index="1"/>

     <output id="Transform_output" datatype="Object" index="0"/>

     <script lang="Python" src="C:\rube\Transform.py" func="Transform"/>
</block>
```

```
def Transform(input,output):
    output[0] = input[0]

    output[0].LocX = input[0].LocX + input[1][0][0]
    output[0].LocY = input[0].LocY + input[1][0][1]
    output[0].LocZ = input[0].LocZ + input[1][0][2]

    output[0].RotX = input[0].RotX + input[1][1][0]
    output[0].RotY = input[0].RotY + input[1][1][1]
    output[0].RotZ = input[0].RotZ + input[1][1][2]

    output[0].SizeX = input[0].SizeX + input[1][2][0]
    output[0].SizeY = input[0].SizeY + input[1][2][1]
    output[0].SizeZ = input[0].SizeZ + input[1][2][2]
```

Figure 3–8. Example of the user–defined Python script code

In the process of translation, which produces an executable simulation code, a program model defined in MXL is translated into another XML modeling language, Dynamic eXchange Language (DXL), which is also developed by the modeling and simulation research group at the University of Florida. While MXL maintains heterogeneous model types and uses different tags for different model types and elements, DXL is a simple modeling language using only block and connection as model components [83]. The final simulation code, which is written in Python, is generated based on the DXL file. When DXL is translated into the actual Python simulation code, pieces of functional codes associated with each operational block/icon are glued together. A simulation package called SimPack [84] is also imported to provide various simulation methods for the model execution.

The XML representation of model components opens the possibility for user–created model components to be used in Web–based modeling and simulation.

Web–based simulation brings the benefits of Web–based technologies into modeling and simulation, such as distributed modeling, easy accessibility, reusability, and platform–independent execution [85, 86]. To provide concrete and robust information of the MXL structure, an MXL schema is defined [87]. The MXL schema defines the internal structure and the contents of an MXL document. The MXL schema definitions for model types and model files are one way of using ontologies in the RUBE framework. Ontology can be used to store both class and instance information about models, model types, or model structures. Creation of ontologies emphasize knowledge and its representation within computer simulation models [88].

3.3.2 Model Execution in RUBE

Models defined using the RUBE framework are executed based on discrete–event scheduling provided by the simulation package, SimPack. When a DXL model file is translated into a simulation Python script, blocks that have no inputs are inserted into a global future event data structure called Future Event List. The execution order of FBM blocks with no inputs is random, and each of these blocks creates a separate execution thread. Once every no–input block is enqueued into Future Event List, the execution of the model is continued as follows [72]:

1. An event is removed from the head of the future event list.
2. The routine corresponding to this event is now in "control."
3. The event routine will schedule other events.

Therefore, after the execution of the block from the head of Future Event List, the next block attached to the block in control is scheduled into Future Event List. The attachment relationship among blocks in the user model can be found from the trace elements in the correspondent MLX file. The executions of each thread generated from no–input blocks are parallel unless they merge at some

point. Multiple outputs of a single block also spawn parallel execution threads. The execution order of dependent blocks is kept as it is defined in the MXL file. Blocks are executed whenever they receive necessary inputs. Blocks that take inputs from more than one block can be executed in synchronous mode or asynchronous mode. The execution mode of each block is defined in DXL and the default mode is synchronous. When the program model is executed in synchronous mode, blocks wait until they receive all the inputs, whereas in asynchronous mode, blocks are executed whenever they receive any single input.

3.4 Modeling Graphics Program

Programming a code to generate computer graphics artifacts usually involves learning a language such as C++ or Java, and then coding in this language while employing underlying application programming interfaces (APIs) such as OpenGL or Java3D. This way of programming is effective and yet the question remains as to why we are not leveraging the power of 3D computer graphics in the programming process itself. This section introduces an approach and implementation to graphics programming that begins with the user constructing a 3D program, followed by the user executing that program to generate 3D graphics. The graphics programming approach introduced in this section is inspired by the fact that beginning authors can construct interesting scenes by combining primitive geometry objects in creative ways . This work is viewed as providing a user–customized method of computer graphics programming, while leveraging the ability to blend both program syntax and semantics within the same virtual space.

The program model which produces a bowling alley is introduced as an example where customized icons and dynamic model types are used to construct a graphics program. In this example, programming principles and elements, such as functions and parameters, control and data flow, branches and loops, and concurrent executions, are modeled by elements and principles in RUBE FBM.

Instead of simple blocks and arrows, which are used in most block diagrams, machines and conveyer belts in a factory are used as icons to represent program elements, such as functions and parameters.

3.4.1 Requirement Specification

As with most software development, the creation of a graphics program in the RUBE framework starts with activities of specification. The functionality and goal of software are defined at this phase. The goal of a graphics program is centered around rendering and animation of 3D geometric objects. Activities of specification for a graphics program include such questions to be rendered as what, how and where. The example software that we want to build using the RUBE framework is a program that generates a 3D bowling alley consisting of a floor, a ball, and pins. We want 3D objects for a floor, a bowl, and four pins to be textured with proper images and located in interesting positions.

3.4.2 Design

Developing a graphics program requires a number of primitive geometric objects to be declared and operations for the manipulation of these objects to be specified. Objects that will be used in our 3D bowling alley example are a cube, a bowling ball, and a bowling pin. Operations necessary for the manipulation of those objects are transformation, texturing, and duplication. Generation and manipulation of 3D objects for a floor, a bowl, and pins are three independent operational groups of texturing and transformation. Independent operational groups do not necessarily need to be programmed in imperative style if the programming language supports concurrent programming. Facilitated by the parallelism in RUBE, as introduced in Section 3.2.2, we will program the three operational groups with three different threads. A generation of four identical pins is programmed by the loop modeling, as discussed in Section 3.2.2. Figure 3–9 shows the block diagram for our example program. First, two execution threads

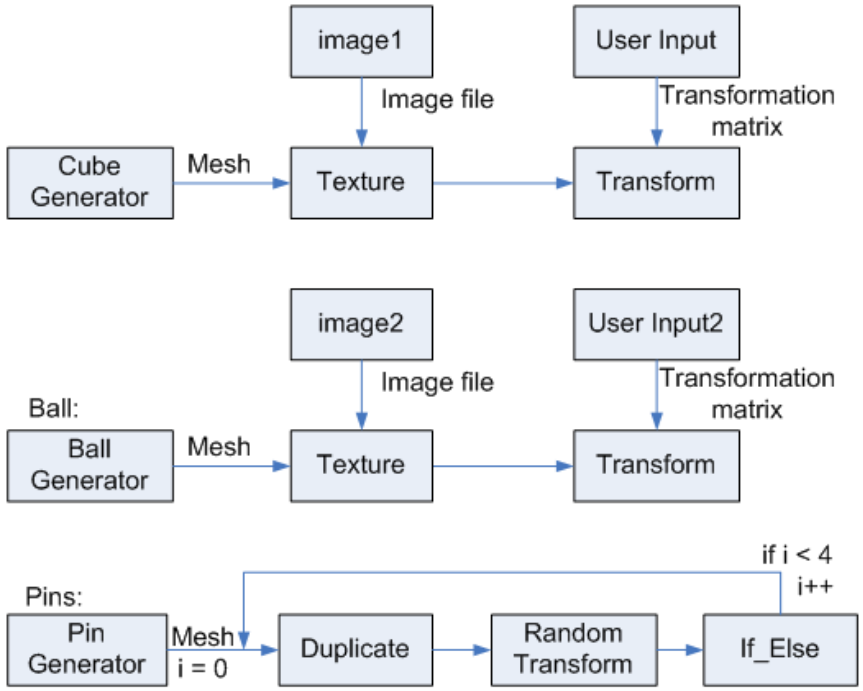

Figure 3–9. FBM for the bowling alley generation

are composed of simple operations, such as texturing and transformation for the generation of a textured floor and ball, respectively. The third execution thread generates multiple textured pins. The circular connection of blocks in this thread models the repeatable execution of Duplicate and Transform operations to produce multiple pins. Once the control moves to the If_Else block, the execution of the thread continues or ceases depending on the current condition of this block. In this example, when the execution is evaluated to be continued, then the If_ Else block produces output to the Duplicate block, otherwise it produces nothing.

The RUBE framework provides an integrative modeling environment in which different model types, such as dynamic models and geometry models, exist within the same 3D space [37]. Using the RUBE framework, it is also possible to juxtapose the 3D output with the source program model. If we plan to integrate the 3D output with the 3D program model, the layout for both should also be designed. Figure 3–10 shows the design of the layout for 3D integrated

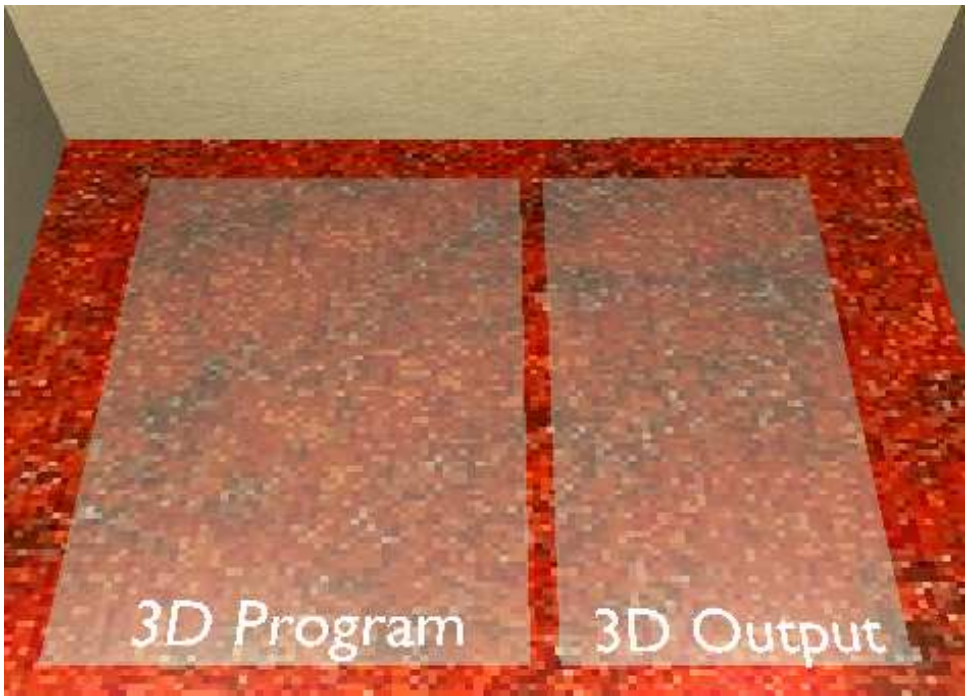

Figure 3–10. Layout design for a 3D integrated model

programming. The 3D program will be placed on the left and the 3D output will be positioned on the right.

3.4.3 Implementation

Implementing a program from the RUBE framework involves developing pieces of Python codes and representative 3D icons for each functional unit of the program. Necessary functional units in the bowling alley program include Cube generator, Bowl generator, Pin generator, Transform, Texture, Duplicate, If_Else, and User Input. With the design discussed in Section 3.4.2, a 3D graphics program for a bowling alley can be constructed using the RUBE framework as follows:

1. Write Python functions: Based on the parameter rules specified in Figure 3–8, pieces of Python codes for functional units can be written using any text editor.

2. Create 3D icons: Users can create their own 3D icons to represent each Python function using any 3D tool if the icons can be exported into a format that Blender can import. Figure 3–11 shows the factory metaphor icons used for the bowling ally program.

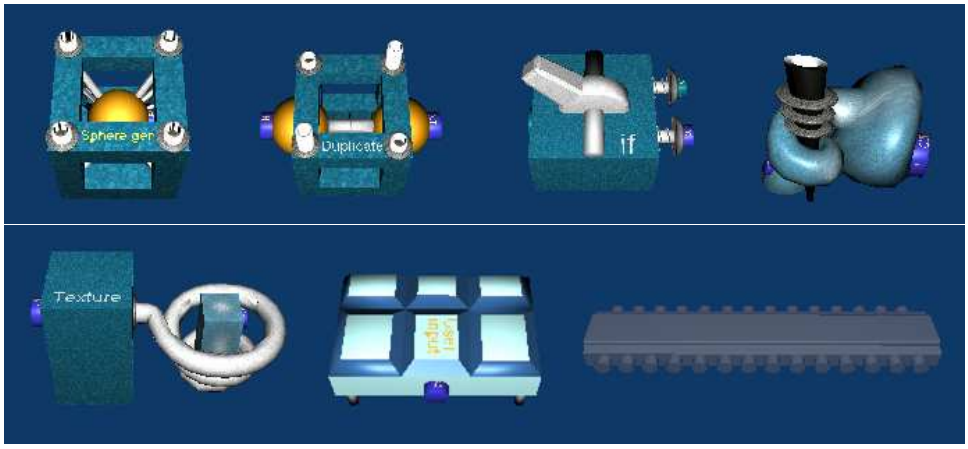

Figure 3–11. Graphics programming icons

3. Insert 1 and 2 into the RUBE framework: Once pieces of Python codes and 3D icons are developed, they are ready to be used in the RUBE framework by placing them into the directory where RUBE is installed. The RUBE directories are structured into three sub–directories–primitive, predefined, and user–defined–for different themes of icons.

4. Open Blender and load the blender interface.

5. Import icons: As you browse the RUBE directories from the blender interface, newly inserted icons along with pre–existing icons will appear as a button. Clicking this button will import the representative 3D icon into the Blender scene and link the associated Python script to the current scene.

6. Connect icons: There are two ways of connecting icons: Either select the source icon and the destination icon together and click the trace object, or simple click the mouse and place the 3D cursor between the source and the destination icons. Figure 3–12 shows the snapshot of connecting two icons. Upon the selection of icons for Cube generator and Duplicate, clicking the Straightbelt button connects two icons with a straight belt icon.

7. Generate Program: After importing necessary functions into the current scene and properly connecting them, clicking the "SIMULATION" button from the blender interface will generate the Python program for the 3D bowling alley graphics.

If we want to use the functions only from the RUBE library, we can skip steps 1 to 3. The RUBE framework provides a function library for basic operations with primitive and predefined icons. Basic mathematical operations (add, subtract,

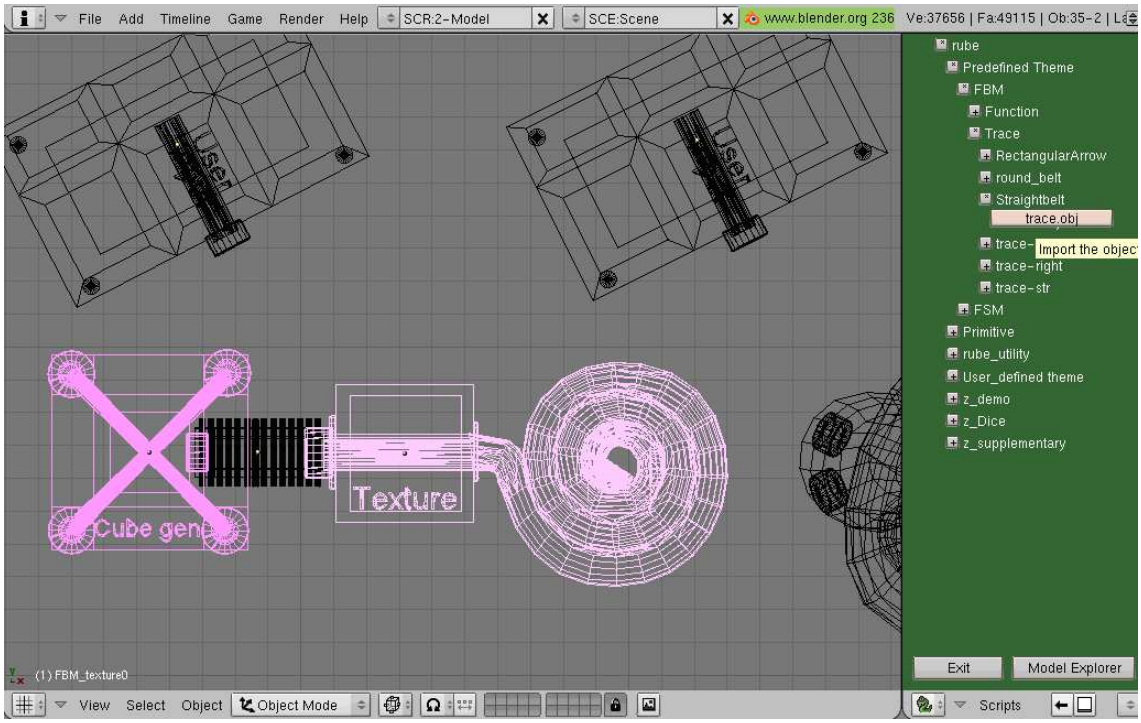

Figure 3–12. Snapshot of connecting icons from the blender interface

multiply, divide, and random number generate) and graphics operations (transform, duplicate, material, and texture) are provided in the RUBE function library. Primitive icons are cubes and spheres. Predefined themes include icons using the factory and the chemistry metaphors.

Figure 3–13 shows the 3D program model built from the blender interface for the bowling alley example using the factory metaphor icons. Machines represent functional units and the conveyer–belts represent the data flow between blocks. The 3D program in Figure 3–13 has the same structure as the block diagram in Figure 3–9. As discussed in Section 3.3.2, blocks with no inputs are executed first in parallel. These are the Cube, the Ball, the Pin generators, and four User Inputs.

3.4.4 Execution and Validation

The executable Python program from Figure 3–13 is generated by clicking the SIMULATION button from the blender interface. Internally, the 3D program is stored in MXL. Through the MXL DXL Python translation chain, an executable

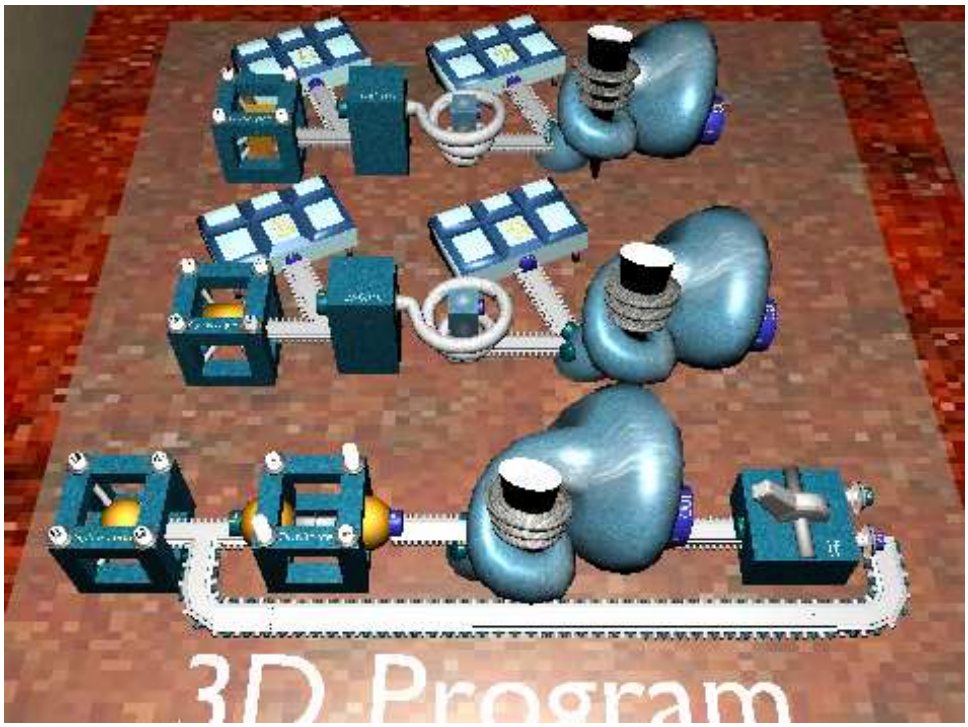

Figure 3–13. Program model for the bowling alley using a factory metaphor

program that generates a 3D bowling alley is created. Execution of the final Python script produces a real–time animation of constructing the 3D bowling alley. It is therefore possible to watch 3D objects being generated and manipulated as the program runs. Also, by inserting a simple recording code to the pieces of the Python code so that the one in the control records the current scene, the entire process of constructing the 3D output is created as a movie. Figure 3–14 is the final output of the 3D bowling alley integrated with the 3D program model. Four snapshots are taken from different angles: a) top view, b) right side view, c) left side view, and d) close–up view of the bowling alley. The 3D program is placed on the left and the resulted 3D bowling alley on the right as designed in Section 3.4.2.

By simply changing the parameters of Texture blocks and the iteration condition in the If_Else block, we could get very different 3D bowling alleys. Figure 3–15 shows two different bowling alleys which are produced from the execution of the program model in Figure 3–13 with different parameters. Since the model

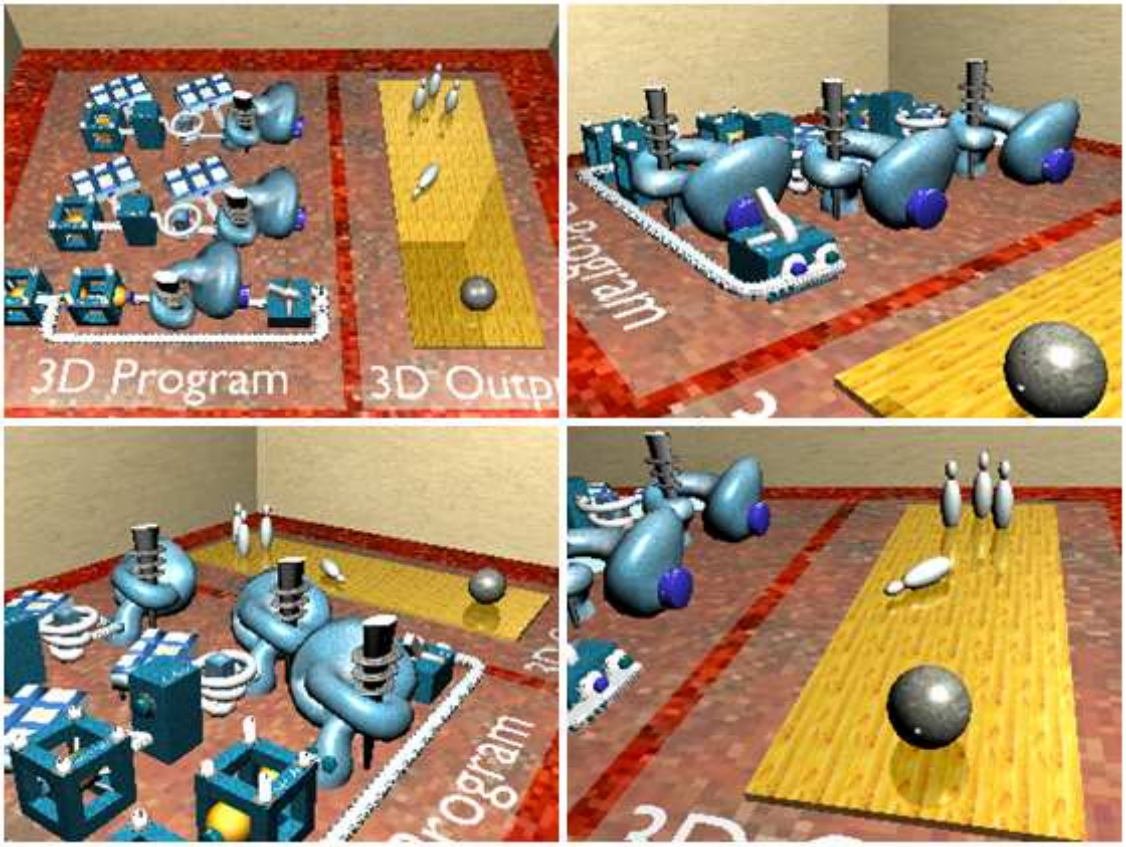

Figure 3–14. Snapshots of program model for the bowling alley using a factory metaphor from different angles: a) top view, b) right side view, c) left side view, and d) close–up view of the bowling alley

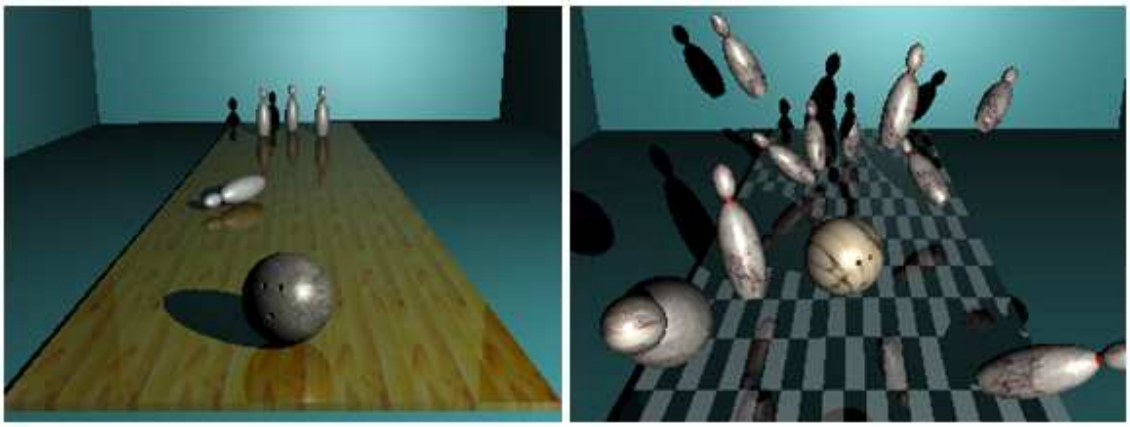

Figure 3–15. 3D bowling alleys generated from the program model in Figure 3–13 with different parameters

```
wihle(currentChar != EOF)
  if(currentChar == '/')
     getNextChar()
        if(currentChar == '*')
           getNextChar()
           do
              while(currentChar != '*')
                 getNextChar()
              getNextChar()
              if(currntChar == '/') break
           while(currentChar != '/')
           return newToken("comment");
     else getNextChar()
  else getNextChar();
```

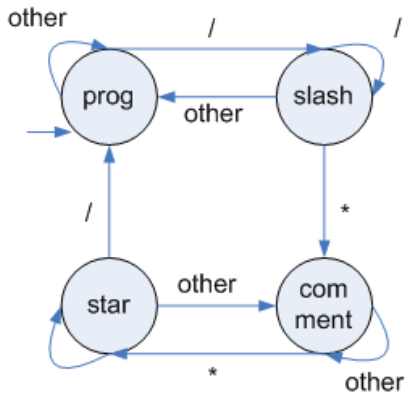

Figure 3–16. Algorithm and Finite State Machine(FSM) for comment scanner

components are distributed in a file structure, different 3D bowling alleys could be generated not by modifying lines of codes from a long program, but by modifying them from the necessary modules and reproducing the output file by running the translation engines. In Figures 14 and 15, Lightning effects such as ray–tracing and shading are added for better graphics.

3.5 Modeling State–based Program

In this section, we will discuss how a comment scanner for C and Java programs is programmed using a chemistry metaphor and multimodeling in RUBE. In C and Java programs, everything between '/*' and '*/' is regarded as a comment and ignored. Thus, the comment scanner for C and Java first looks for '/' followed by '*'. When the scanner finds '/*', every character that follows is ignored until the scanner finds '*' followed by '/'. Figure 3–16 shows the algorithm and the state diagram for a comment scanner for C and Java. While executing, this algorithm can be one of four states: read a character other than '/'; read '/' and wait for '*'; read '*' following '/'; and read '*' and wait'/'. These four states can be represented by FSM, as described in the right figure of Figure 3–16. It is apparent that the state diagram is more intuitive and easier to understand than the textual algorithm.

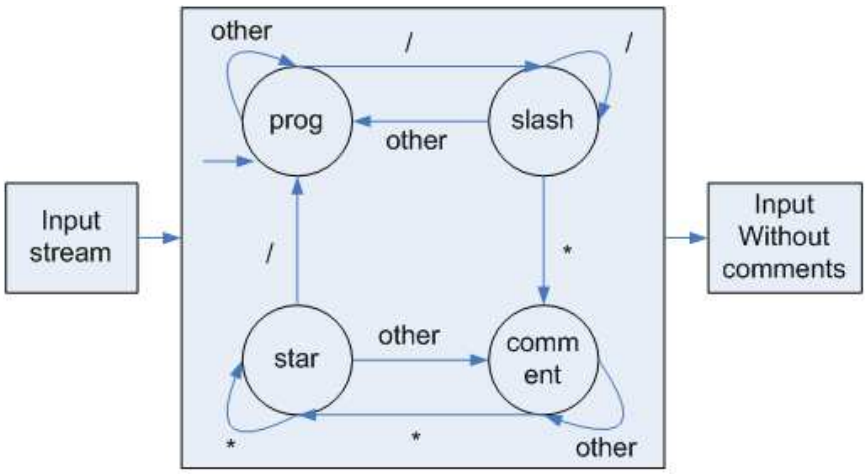

Figure 3–17. RUBE FSM diagram for comment scanner

The Finite State Machine (FSM) provided in RUBE using the blender interface lets users construct a state–based program, such as our comment scanner. Using the blender interface, a comment scanner can be implemented as a 3D state machine, not as a sequential program with frequent conditional branches. Once users develop the necessary piece of codes for each state of the comment scanner and insert them into the RUBE framework, the comment scanner FSM, which operates as described in Figure 3–16, can be constructed. The FSM in RUBE is constructed by importing predefined state icons into 3D Blender space and setting transitions between the state icons, along with the transition conditions using the blender interface. Due to the nature of FSM that requires input and output sets for its execution, FSM in RUBE is defined as a submodel inside of an FBM block, and it forms a multimodel–a model that is composed of other models [82]. The parent FBM provides necessary inputs and takes outputs from FSM enclosed. Figure 3–17 shows the multimodel diagram of FSM for the comment scanner. In this figure, the FSM is placed inside the second block. The first block produces necessary inputs, and the third block takes output of FSM in the second block, which is a program without comments.

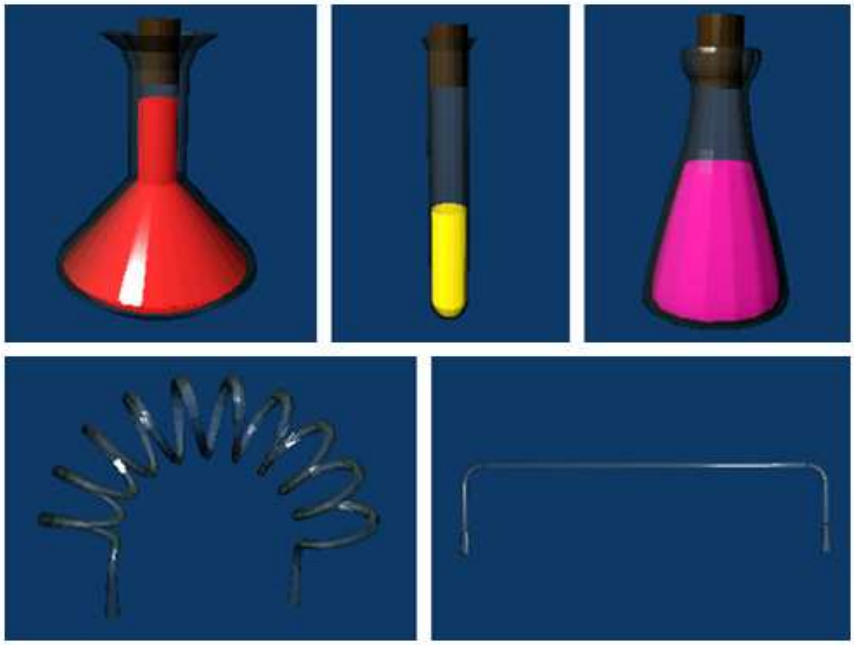

Figure 3–18. Icons for comment scanner

We thought an experimental setting of several chemistry flasks connected by glass tubes is similar to a program which have several states connected by transitions. In this case, the chemistry flasks and glass tubes can represent states and transitions in a state–based program. Figure 3–18 shows the icons for the comment scanner FSM. The multimodel in RUBE is constructed based on the containment relationship between model components: placing a block or state inside of another block or state in the 3D Blender environment.

Figure 3–19 shows the multimodel constructed from the blender interface for the comment scanner in Figure 3–17. In this figure, four chemistry containers are placed inside of the second block and constructing multimodel. The blender interface checks the bounding–boxes of every icon to check the containment relationship between icons and the construct multimodel. Separate model files, MXL files, are generated for models at different nesting levels. Two separate MXL files are generated from Figure 3–19: one for FBM and one for FSM.

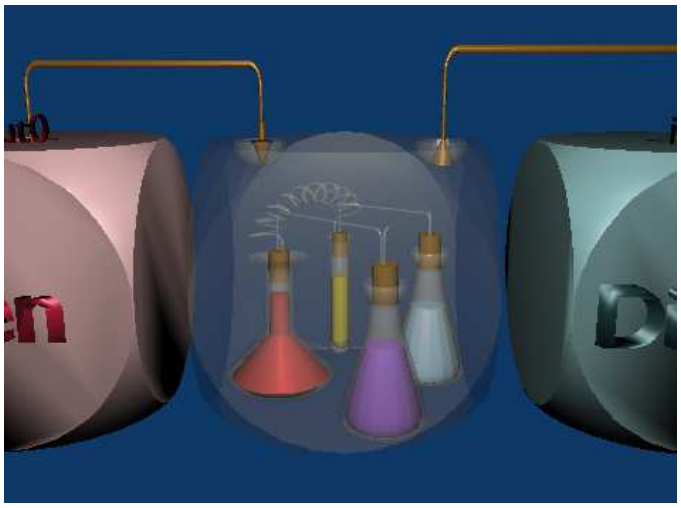

Figure 3–19. Program model for comment scanner using a chemistry metaphor

Visualization of the model execution in RUBE is achieved by adding a simple graphics manipulation code to the Python code for each correspondent state. By inserting codes that generate different 3D graphics for each correspondent state, we can get representative 3D graphics that result from the execution of the program model. Figure 3–20 shows one example of 3D graphics that represents the execution result of the program model in Figure 3–19. Activation of different states generates different bubbles at random–but restricted–locations: Bubbles, which represent comments, are placed at the right–hand side, and bubbles, which represent normal program constructors, are placed at the left–hand side. In Figure 3–20, a lightning effect, such as ray–tracing and shading, are added for better graphics.

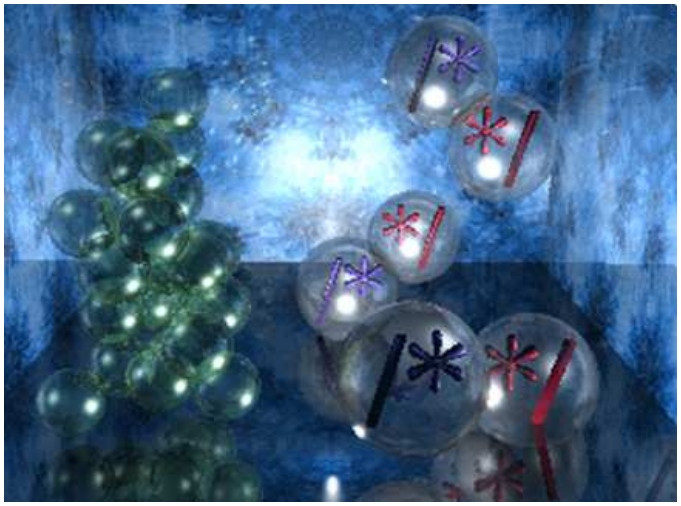

Figure 3–20. 3D graphics generated from the program model in Figure 3–19

CHAPTER 4
INTERACTIVE AND INTEGRATIVE APPROACH TO MODEL AND BEHAVIOR

Within the simulation community, a need exists to create environments for *integrative modeling*, which makes an interactive environment. In this environment, models coexist within the same virtual space with abilities to cross–link between model components [37, 89]. This chapter leverages the integrative approach introduced in Chapter 3 for simulation modeling, especially for a system dynamics modeling. The system dynamics modeling provides a fluid metaphor for representing material flows in systems with feedbacks, otherwise represented using a set of equations with a traditional modeling approach. Our purpose is to demonstrate a new method for modeling the particular type of a predator–prey system by leveraging the power of human interaction, as applied to two different models: a system dynamics model and the population geometry model. Both models are represented in three dimensions at the same time, and they are fully integrated so that interacting with one model's components provides context for the other model's components through a method of highlighting. Clicking on the object representing the rabbit population therefore highlights the rabbit level in the system dynamics graph and vice versa. The models are animated to reflect the phenomenon of the system, which is the change in populations of the predator and the prey. With the integrative modeling approach, the linkages between the related objects from different model types are visualized by effective interaction means.

This approach has the following benefits:

- The use of three dimensions affords a more literal interpretation of the underlying hydraulic metaphor for system dynamics. Levels look like tanks, and the current level looks like water rising and falling. In this sense, the system dynamics model is reminiscent of earlier analog computers except that the environment is virtual rather than physical.

- Both models (dynamics and geometry) are available to the ecologist in a single virtual space of the same spatial dimensionality.

- The dual interaction between both models is a form of integrative modeling where interrelationships among model components are surfaced in the user interface, allowing free form exploration of both models and their equivalences.

- The implementation is built on top of a game engine, demonstrating that the system dynamics approach to ecosystems can successfully use this technology in modeling and simulation.

4.1 Modeling Predator–prey System

A system dynamics model is generally composed of a directed network of rates, levels, auxiliary variables, and constants. Modeling with system dynamics begins with a rough design for a system and gradually proceeds to a computer simulation model. The main focus of system dynamics modeling is to identify feedback loops within the system being modeled. It is a causal graph that captures the causes of dynamics and represents the feedback loops. The rough design represented in a causal graph is then converted into a system dynamics flow graph. With software that can interpret a flow graph, we can simulate the model. Otherwise, a flow graph is further translated into a set of differential equations or system dynamics programming language, DYNAMO, for computer simulation. This section introduces the creation of the causal graph for the predator–prey model, and converting the causal graph into the flow graph for simulation.

We use the fox–rabbit population as an example of the Lotka–Volterra model for system dynamics modeling. Since our purpose is to demonstrate the novel modeling environment, our scenario is simplified with the following three assumptions: 1) only two species exist: fox and rabbit; 2) rabbits are born and then

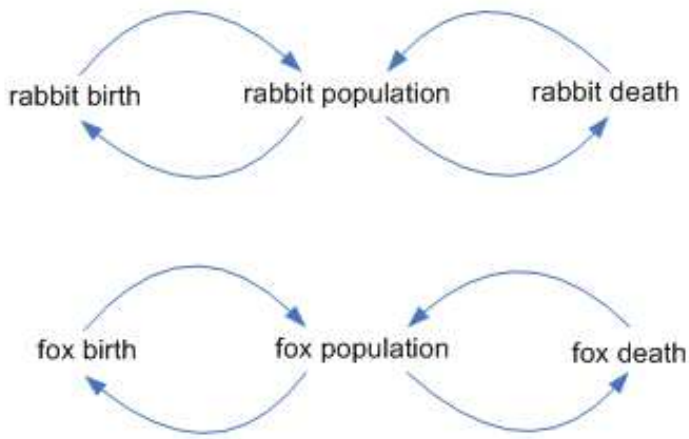

Figure 4–1. Causal graph for rabbit and fox populations

die through predation or inherent death; 3) foxes are born and their birth rate is positively affected by the rate of predation, and they die naturally.

4.1.1 Causal Graph

A causal graph is composed of variables connected by arrows representing the causal influences among variables. As mentioned earlier, causal graphs represent the feedback structure of a system. We need to closely examine the dynamics of the fox–rabbit population and draw a causal graph. We will start by looking at the population dynamics when they are isolated, which means no predation. For both species, we assume there is unlimited food and no threat of death other than inherent death. In this situation, the population dynamics of both species have the same pattern. Their population increases as their birth rate increases and decreases, as their death rate increases. As their population increases, both their birth rate and death rate increase. Their population size has a reinforcing relation with their birth rate and a compensating relation with their death rate. Figure 4–1 shows the causal graph of this model.

Let us now extend our model to the situation where two species co–exist and interact with each other. In this new situation, we assume that while rabbits have

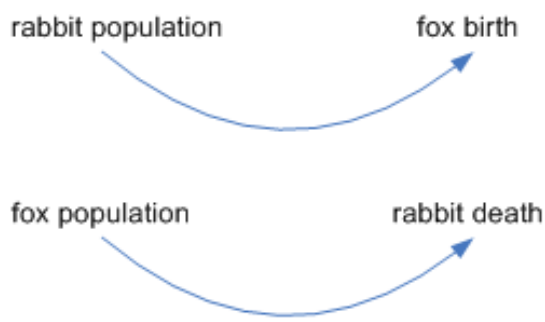

Figure 4–2. Causal graph for fox–rabbit predation

an unlimited food supply, foxes have the limited food supply, which are rabbits. It is also assumed that while the only cause of fox death is natural death, rabbit death is controlled by their inherent death and predation by foxes. While this model is too simplified to represent the real phenomenon of the predator–prey system, it should be a viable example for system dynamics modeling. We can draw a causal graph for the predator–prey model of foxes and rabbits by combining the causal graphs in Figure 4–1 and adding new causal influences representing predation. The two causal influences due to predation are 1) the growth of the fox population increasing rabbit death and 2) the growth of the rabbit population increasing fox birth. Figure 4–2 shows the causal graph for predation.

At this time, we will consider coefficients for birth and death, which are the birth rate and death rate. Although these rates are not a part of the feedback structure for our predator–prey system, they are important factors for simulation. Figure 4–3 shows the expanded causal graph for our predator–prey model with predation and birth and death rates.

While Figure 4–3 captures the feedback structure of our model, it is not sufficient to explain details about the model. To add more information, we can augment the causal graph by signing or labeling the arc and the loop with

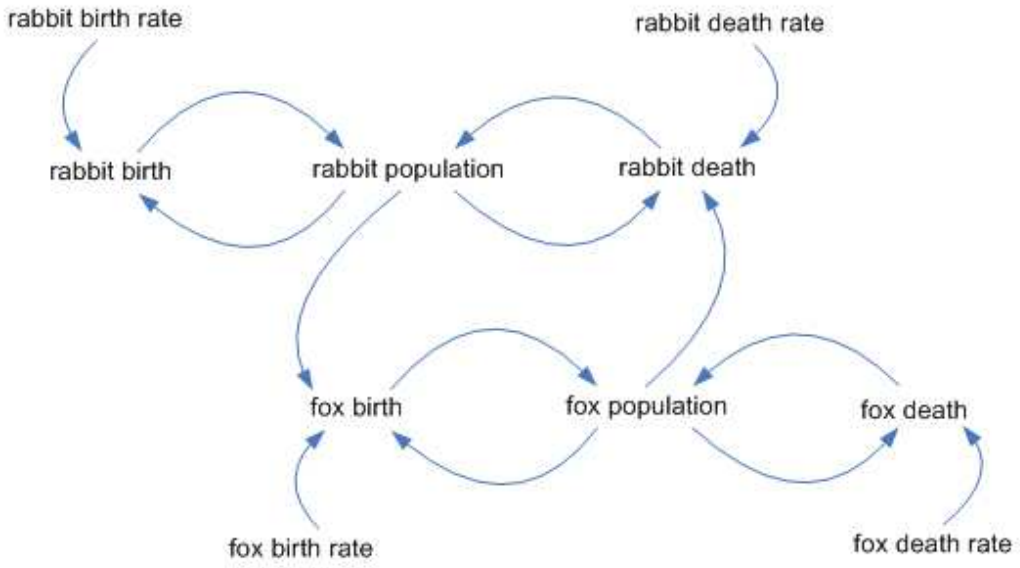

Figure 4–3. Causal graph for the prey–predator model

polarities. Signing an arc is adding a "+" or "–" sign at an arrowhead of each arc. A "+" sign is used for the reinforcing relation. In the reinforcing relation, the effect increases and decreases as the cause increases and decreases, respectively. A "–" sign is used for the compensating relation. In the compensating relation, the effect increases and decreases as the cause decreases and increases, respectively. Once all arcs are labeled, we can determine the polarity of feedback loops. Since the behavior of a system is explained mostly in terms of feedback loops, understanding the characteristics of feedback loops is important. We can predict the dynamics of the feedback loops based on their polarities. The positive feedback loops have a reinforcing effect so that when some quantity of its element increases, it generates behaviors of growth or amplification. The negative feedback loops tend to produce stable, compensating, or goal–seeking behaviors over a period of time. The fast way to determine the polarity of a loop is by counting the number of "–" signs contained within the loop. A feedback loop is positive if it has an even number of

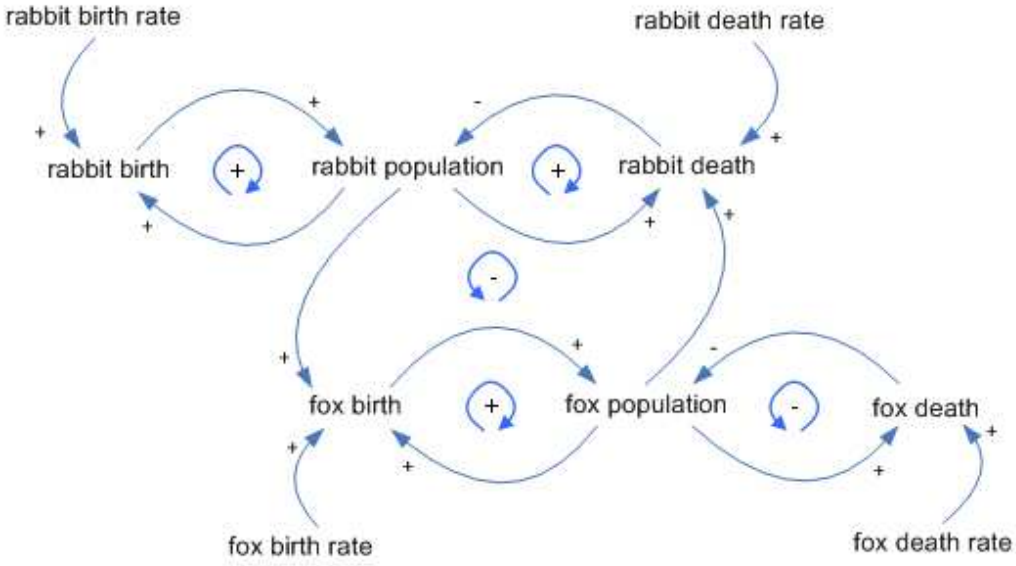

Figure 4–4. Augmented causal graph

arcs with "–" signs, and it is negative if it has an odd number. Figure 4–4 shows the augmented causal graph of Figure 4–3.

4.1.2 Flow Graph

While a causal graph captures the feedback structure of the system and provides insight into the system's behavior, we cannot drive a simulation model from it. To construct a simulation model, a causal graph is converted into a system dynamics flow graph. In this chapter, we will focus on constructing a flow graph from the causal graph introduced in the previous section. Converting a causal graph into a flow graph starts with identifying levels and rates. Levels and rates, along with feedback loops, are the fundamental concepts to represent the dynamic structure of systems with feedback. When identifying levels and rates from a causal graph, it is helpful to consider what variables accumulate over a period of time. Levels are a quantity or accumulator that amass over a period of time, while rates are an activity, movement, or flow that changes the value of levels. We can think of

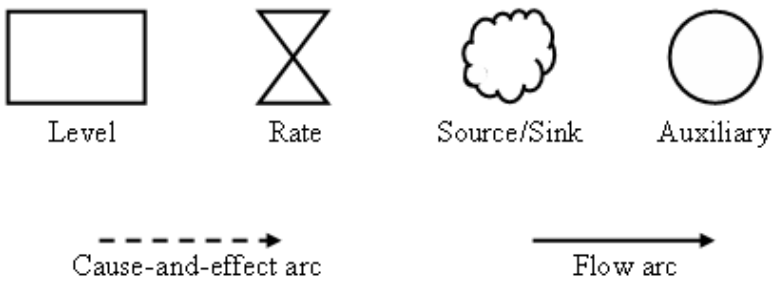

Figure 4–5. System dynamics flow graph symbols

levels and rates in terms of a fluidic metaphor where water valves can be adjusted to change the amount of water in water tanks, thus providing insight into their concepts. In this case, water valves are a rate and water tanks are a level. Levels and rates outside of the system boundary are modeled by a source or a sink in system dynamics modeling. A source is where the flow of material starts, and a sink is where it terminates outside the system. Although levels and rates are enough to describe the system mathematically, auxiliary variables are used for easy communication and clarity when a level or a rate involves complex equations. Auxiliary variables are useful, especially when formulating the influence of a rate involving more than one intermediate calculation. While a causal graph does not differentiate the flow of material from the cause–and–effect relation, it is modeled as different elements within a flow graph. A flow from one level to another is modeled by a flow arc, and the influence on a rate by other elements is represented by a cause–and–effect arc. Figure 4–5 shows the symbols for the elements of a flow graph.

We are now ready to convert the causal graph in Figure 4–4 into a flow graph using the elements in Figure 4–5. As mentioned earlier, the conversion starts with identifying levels and rates. In our model, two quantities change their values over a period of time: the fox population and the rabbit population. These quantities are converted into levels. Birth and death are converted into rates since they change

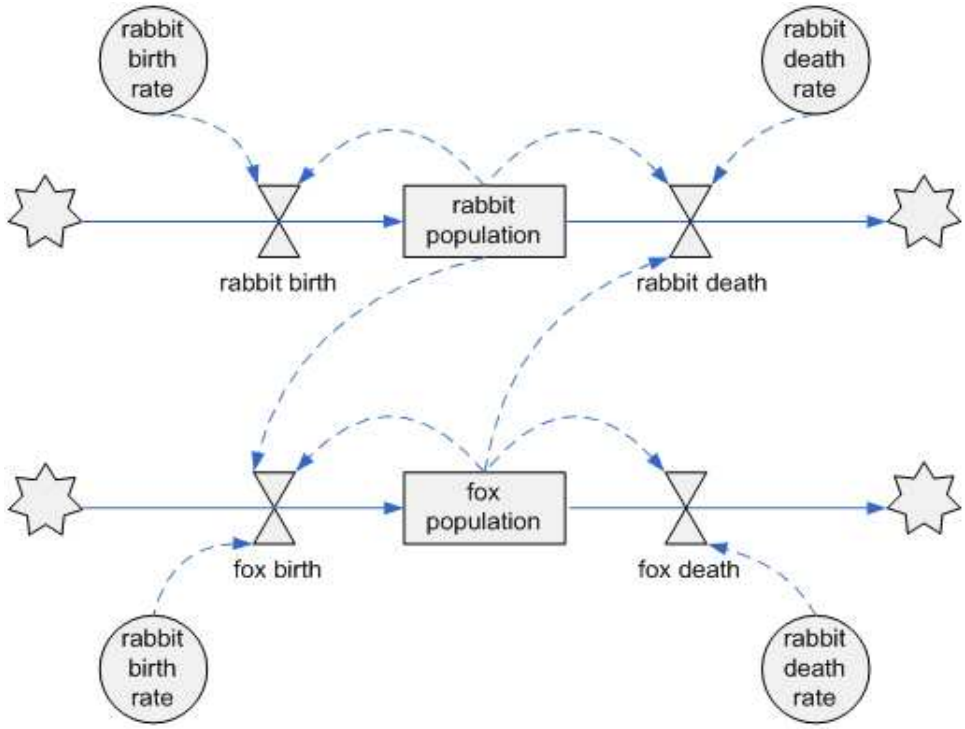

Figure 4–6. System dynamics flow graph for the causal graph in Figure 4–4

the size of the population. For both species, other than their own populations, birth and death are influenced by the rate of birth and death, respectively. The birth rate and the death rate are therefore converted into auxiliary variables. Once every variable in the causal graph is converted, the arcs are considered. As the arcs going from a rate to a level represent material flows, they are modeled by flow arcs. The rest of the arcs, which represent causal influences, are converted into cause–and–effect arcs. The resulting flow graph of these conversions is shown in Figure 4–6.

4.1.3 Simulation

Visual tools, such as Vensim [90] and STELLA [91], allow users to build and simulate causal graphs and flow graphs for dynamic systems. With Vensim, building causal graphs and flow graphs is achieved by drawing a diagram from its graphics editor. The equation editor in Vensim helps users to build a simulation model using the causal influences represented in the flow graph. Figure 4–7

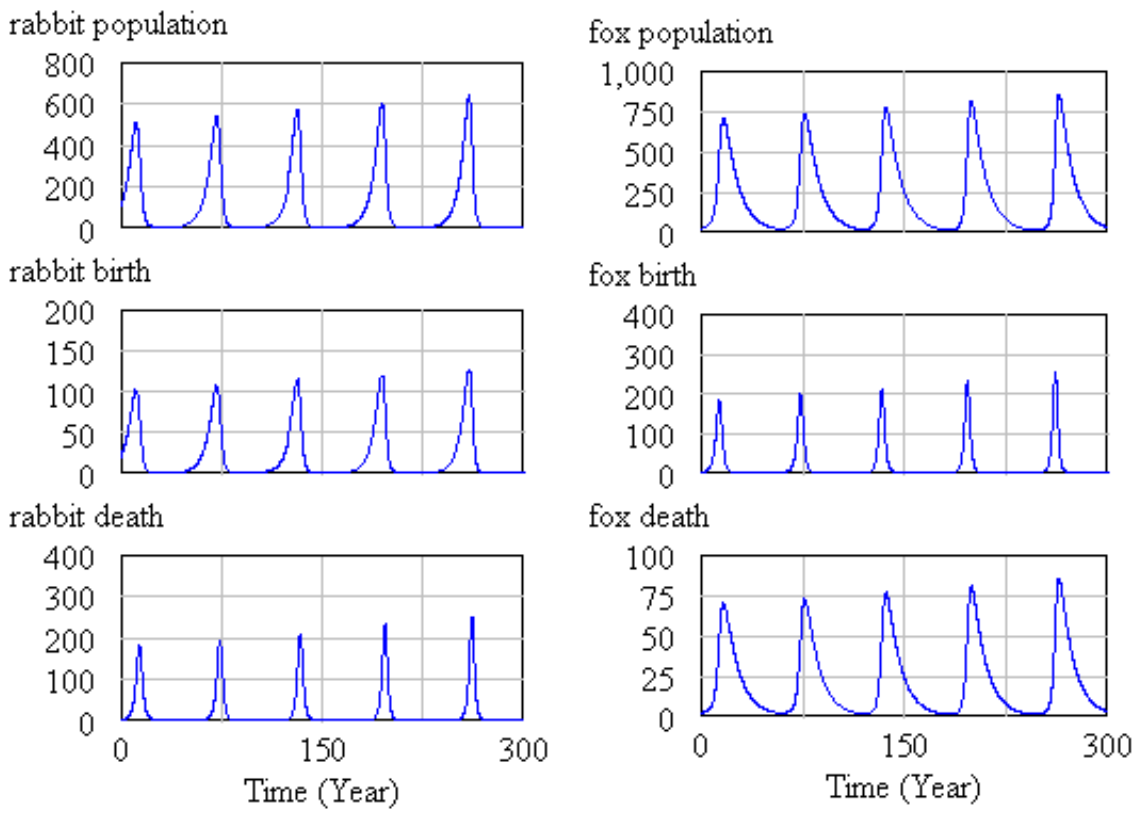

Figure 4–7. Vensim simulation result of the predator–prey model

shows the simulation result graphs using Vensim for our predator–prey model, respectively.

We can convert a flow graph into a set of equations using the algorithm shown in Table 4–1. Let us convert our model in Figure 4–6 into a set of equations by applying this algorithm. The first part or paragraph of this algorithm is the process of augmenting the causal graph and converting it into the system dynamics flow graph, which we already did in Figures 4–3, 4–4, and 4–6. Let us proceed to the second part or paragraph of this algorithm, which is writing differential equations for each level. In our model, we have two levels: the rabbit population and the fox population. If we let R be the rabbit population and F be the fox population, we

Table 4–1. Algorithm for converting a causal graph into a set of differential equations

```
Program Main
     We are given a concept graph with modes and arcs
          The arcs require sign (+,-) labeling
          The nodes require labeling:
               source, rate, level, constants, auxiliary
     For each level node (L) with an input rage node (R1)
          and an output rate node (R2) write:
          dL/dt = k1*R1 - k2*R2
          where k1 and k2 are rate constants
     End For
     For all other nodes (N) write:
          N(t) = a linear function of all inset members of this node
     EndFor
EndMain
```

get the following equations:

$$dR/dt = rabbit\ birth - rabbit\ death$$

$$dF/dt = fox\ birth - fox\ death$$

We now proceed to the third part of the algorithm, which is defining linear functions for other nodes as follows:

$$rabbit\ birth = rabbit\ birth\ rate * R$$

$$rabbit\ death = rabbit\ death\ rate * R * F$$

$$fox\ birth = fox\ birth\ rate * R * F$$

$$fox\ death = fox\ death\ rate * F$$

Substituting the terms in the right-hand side of dR/dt and dR/dt with these linear functions produces the following equations which are the tradition equation

models of the predator-prey system:

$$\begin{aligned} dR/dt &= rabbit\ birth\ rate * R - rabbit\ death\ rate * R * F \\ dF/dt &= fox\ birth\ rate * R * F - fox\ death\ rate * F \end{aligned}$$

4.2 Model Visualization

So far, we have examined the computational perspectives of our predator–prey model. In this section, we focus on model visualization. The model visualization can be achieved by various methods from pencil–and–paper drawings to 2D and 3D computer graphics. Although high quality 2D visualizations can show perspective and depth, navigating the image is normally not allowed. With the availability of low cost and high performance hardware for graphics processing, we can expand the model visualization into a more realistic and interesting level into three dimensions. The 3D representation is extremely expressive, and it offers a significant number of degrees of freedom in visualization that is not present in two dimensions, or at least extending what is available into two dimensions. The degree of freedom in 3D visualization is enhanced by navigation, interaction, and various rendering effects, such as texturing and lighting. With 3D graphics, viewers can look at the scene from several interesting viewpoints, navigate effectively from place to place, and interact with the scene objects to examine or trigger interesting behaviors. Such benefits of 3D graphics can bring a feeling of presence and immersion for viewers, resulting in increased engagement with models presented in the visualization.

4.2.1 Model Integration

This section demonstrates a new integrative method for modeling where different model types of a system dynamics model and a population geometry model are represented and are fully integrated in the same virtual space. Traditionally, visualizing model geometry and dynamics is often regarded as

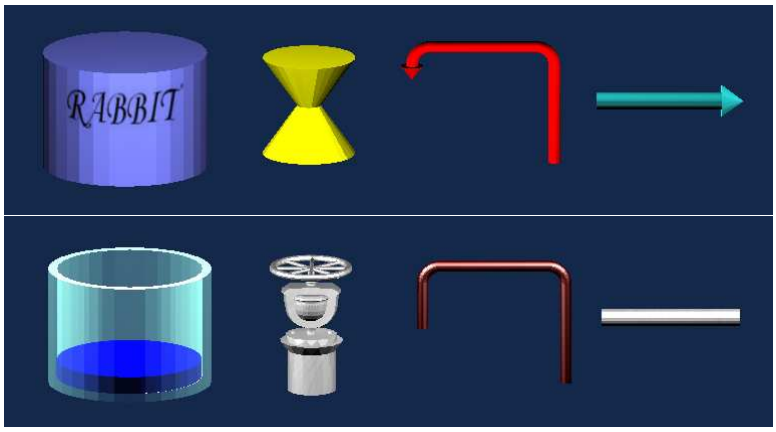

Figure 4–8. System dynamics 3D symbols: From left, level, rate, cause–and–effect arc, and flow arc

two different areas, and little connection or interaction exists between these models. With textural or 2D modeling, the dynamic models that capture the phenomenon of the system are represented by a set of equations or diagrams. Tables and graphs are typically used for displaying the simulation results. In Section 4.1.2, we used 2D flow graphs to represent the dynamics of our model and used graphs to show the simulation results of our model. With this approach to modeling, the focus is fixed on a specific perspective of the model such, as simulation results.

The main focus of an integrative approach to a predator–prey system is on visualizing the dynamic model shown in Figure 4–6, along with its phenomenon or geometry model. For visualizing Figure 4–6 in 3D, we can simply replace the 2D symbols of the flow graph with their 3D counterpart objects. We can also use different metaphoric or customized objects for each model element. The 3D icons for system dynamics symbols are shown in Figure 4–8.

Figure 4–9 shows two different 3D representations of our predator–prey flow graph. In the figure on the left, the levels, rates, and arcs are simply replaced by 3D objects for their 2D notations. The figure on the right uses a fluidic metaphor where levels, rates, and arcs are represented by faucets, water tanks, and water pipes, respectively. The use of three dimensions affords a more literal interpretation

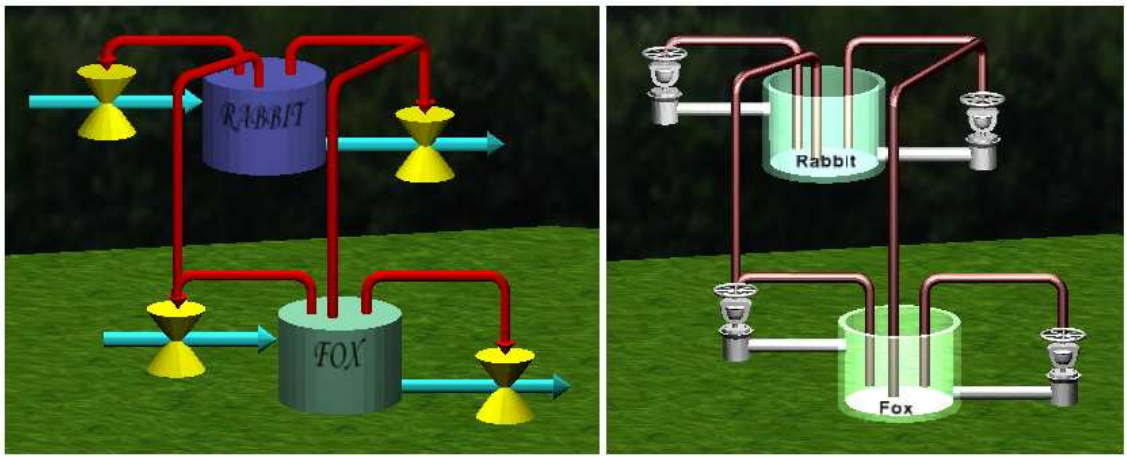

Figure 4–9. 3D representations of the predator–prey flow graph

Figure 4–10. 3D geometry models for the rabbit and fox

of the underlying hydraulic metaphor for system dynamics. Levels look like tanks, and the current level looks like water rising and falling a the tank.

Cylinders and the glass water tanks in Figure 4–9 represent the fox and the rabbit. However, with a integrative modeling approach in which different model types are presented within the same scene, we are also interested in geometry model objects within our model visualization. We therefore include the geometry model shown in Figure 4–10 in the model visualization.

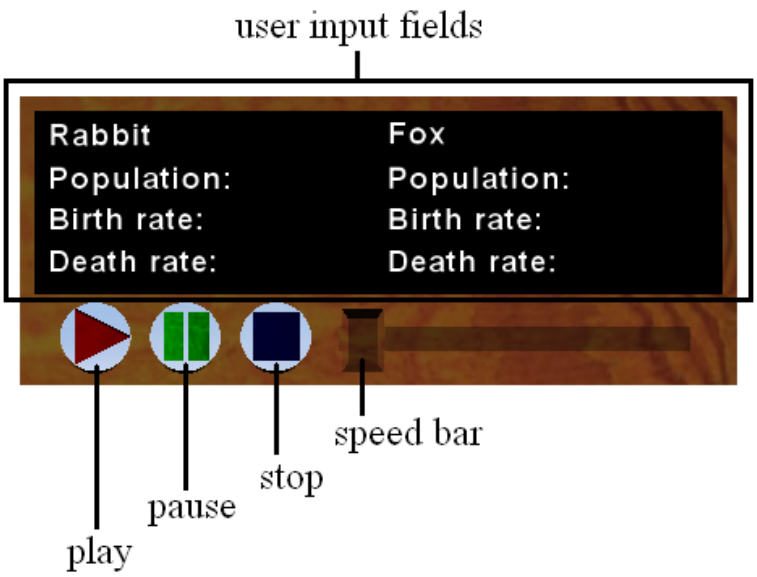

Figure 4–11. Control panel for model simulation

In the model scene, we further integrate the interface for setting simulation variables. Figure 4–11 shows the main control panel for simulation. White colored strings written on the blackboard are modeling variables of the predator–prey system. Clicking on these white strings lets users set the values for these variables. The icons at the bottom of the control panel provide interface for controlling simulation. The red triangle, two green vertical bars, and the dark blue box are the buttons to do the play–pause–and–stop simulation. The sliding bar next to these three buttons is the speed bar that controls the speed of animation. Using these buttons, users can examine the model with various options. For example, users can stop simulation at any point to change the execution speed or to examine the model.

The integrated model for our predator–prey system is shown in Figure 4–12. In this model, extraneous modeling elements, such as grass, ground, forest, and sky, are included. They are not part of the dynamic model, but, as part of the geometry model, they add realism to our visualization. The two spheres at the bottom left of Figure 4–12, enclosing the blue marble and the flow graph symbols, are icons

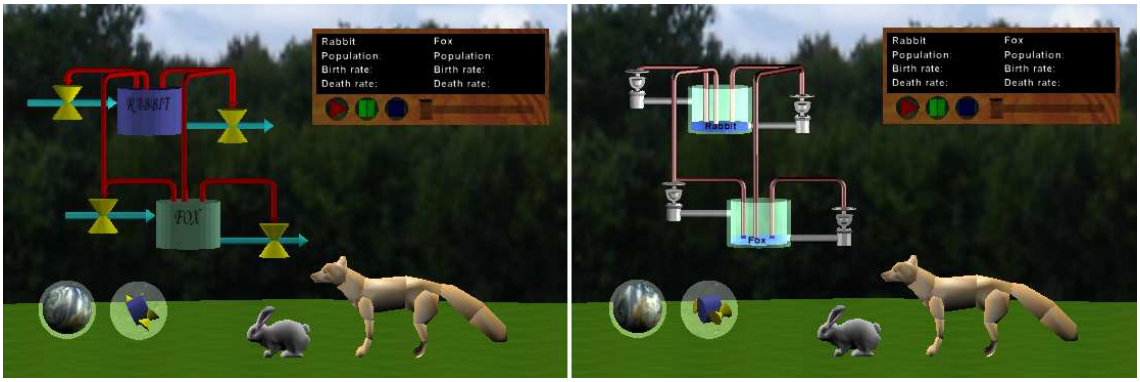

Figure 4–12. Integrated model for the predator–prey system

that toggle the representations of dynamic models between the flow graph and the fluidic metaphor. The left figure shows the model when the flow graph view is selected, and the figure at the right shows when the fluidic metaphoric view is selected.

4.2.2 Model Interaction and Execution

So far, we have examined the static perspectives of our model visualization. Now we need to consider the models dynamic aspects. Dynamic aspects refer to interaction and execution of visualization.

GameBlender. The basic framework for the interaction and execution of the predator–prey model is GameBlender [92]. GameBlender is an integrated environment for making games. It provides basic mechanisms to handle interactions between the user and the scene. The game logic in GameBlender is assembled in the real–time buttons. Users wire different logic bricks together to make a game. The three types of logic bricks are Sensors, Controllers, and Actuators. Sensors act like receptors. They sense events from the mouse, keyboard, joy stick, collision, message, and so forth. Controllers act as the brain for the game logic. The controllers can be very simple expressions or complex Python scripts. Actuators are muscles of the game engine. They provide a basic mechanism to

execute or animate the game scene. Using Sensors, Controllers, and Actuators of GameBlender, we can build 3D games, ranging from simple to very complex.

Component highlighting. In our approach to modeling, the power of human interaction is applied to two different model types so that interaction with one model's components provides context for the other model's components through a method of visual highlighting. With the integrative modeling approach, we can achieve an explicit component identification for related objects via visual means. The explicit component identification supports the understanding the mappings between different model types presented in the same visualization. In our visualization, the linkage between the related elements is represented by simultaneous highlighting. For example, clicking on the object representing the rabbit population highlights the rabbit geometry model, along with the rabbit level in the system dynamics graph and vice versa. Figure 4–13 shows the simultaneous highlighting of corresponding elements between the dynamics model and the geometry model captured during runtime. The dual interaction between both models provides intuitive understanding of the linkage among the dynamics model, the geometry model, and the phenomenon by capturing them all within the same virtual space. With the integrated visualization, the effort for context switching among different model types is minimized.

Setting values for system variables. Simulating the predator–prey model starts by setting initial values for the system variables. The variables appearing on the blackboard of the control panel are the system variables that must be set before simulation starts. As mentioned earlier, clicking on those strings lets users specify their values. Figure 4–14 shows setting simulation variables. In this figure, all the values except the Fox Death Rate are set. The Fox Death Rate is clicked and highlighted with a pink square. The corresponding model element, which is the faucet located at the bottom right of the dynamics model, is also highlighted.

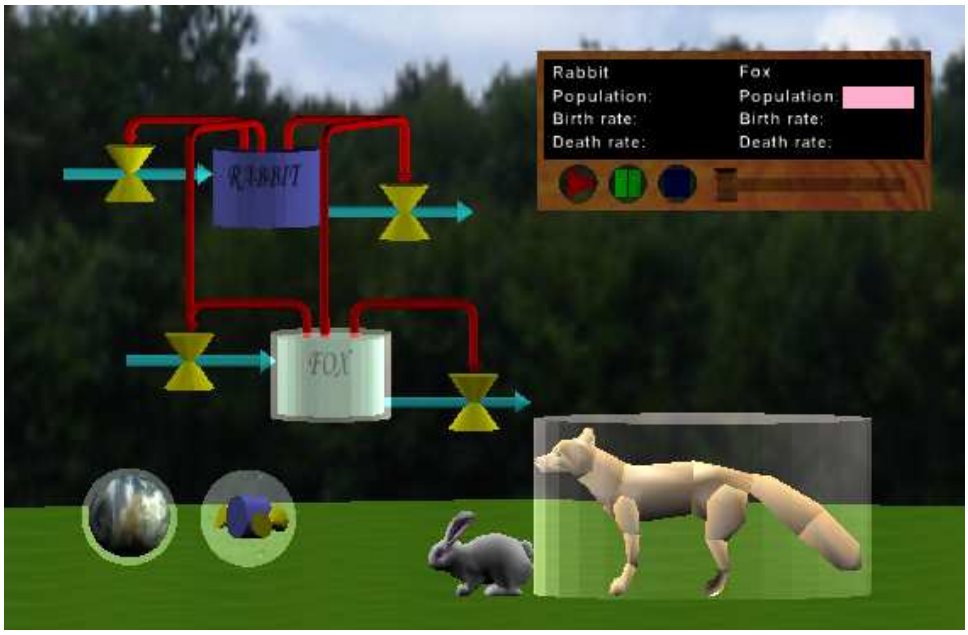

Figure 4–13. Model component identification through highlighting

Users can change values of system variables, such as rates of birth or death, during runtime without leaving the model scene.

Scene animation. We can add dynamics and interactive features to our fox–rabbit population model by means of animation. The size of the fox and the rabbit is changed as their population changes during the simulation. The water tanks, which represent the population of the fox and the rabbit, are filled with virtual water, and the water levels are changed as populations change. The figures in Figure 4–15 are the snapshots taken during the simulation of our integrative model. The water tanks, which are located at the top left of each figure, contain water proportional to the population size of the foxes and rabbits. The size of the foxes and rabbits in the scene is growing and shrinking, as their population increases and decreases, respectively. Numbers next to each system variable, as presented on in the blackboard, show the exact values of each system variable at the point of simulation.

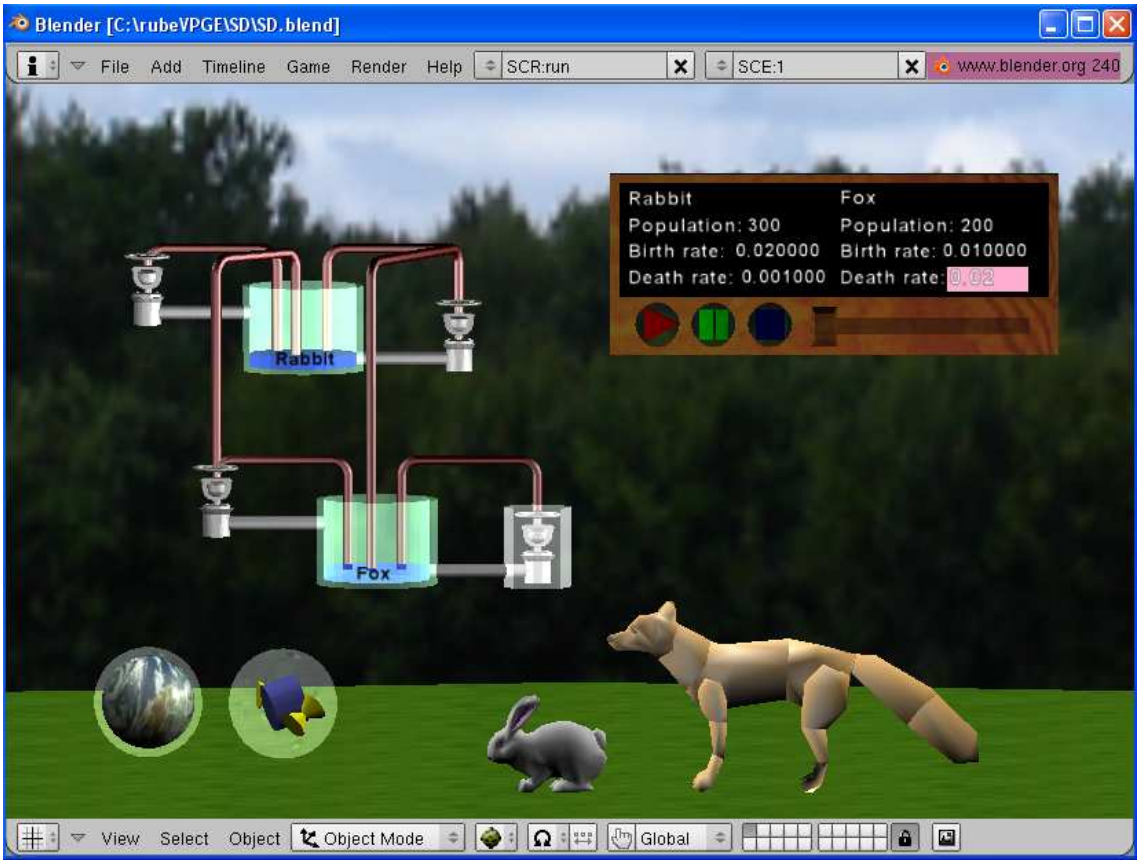

Figure 4–14. Setting simulation variables

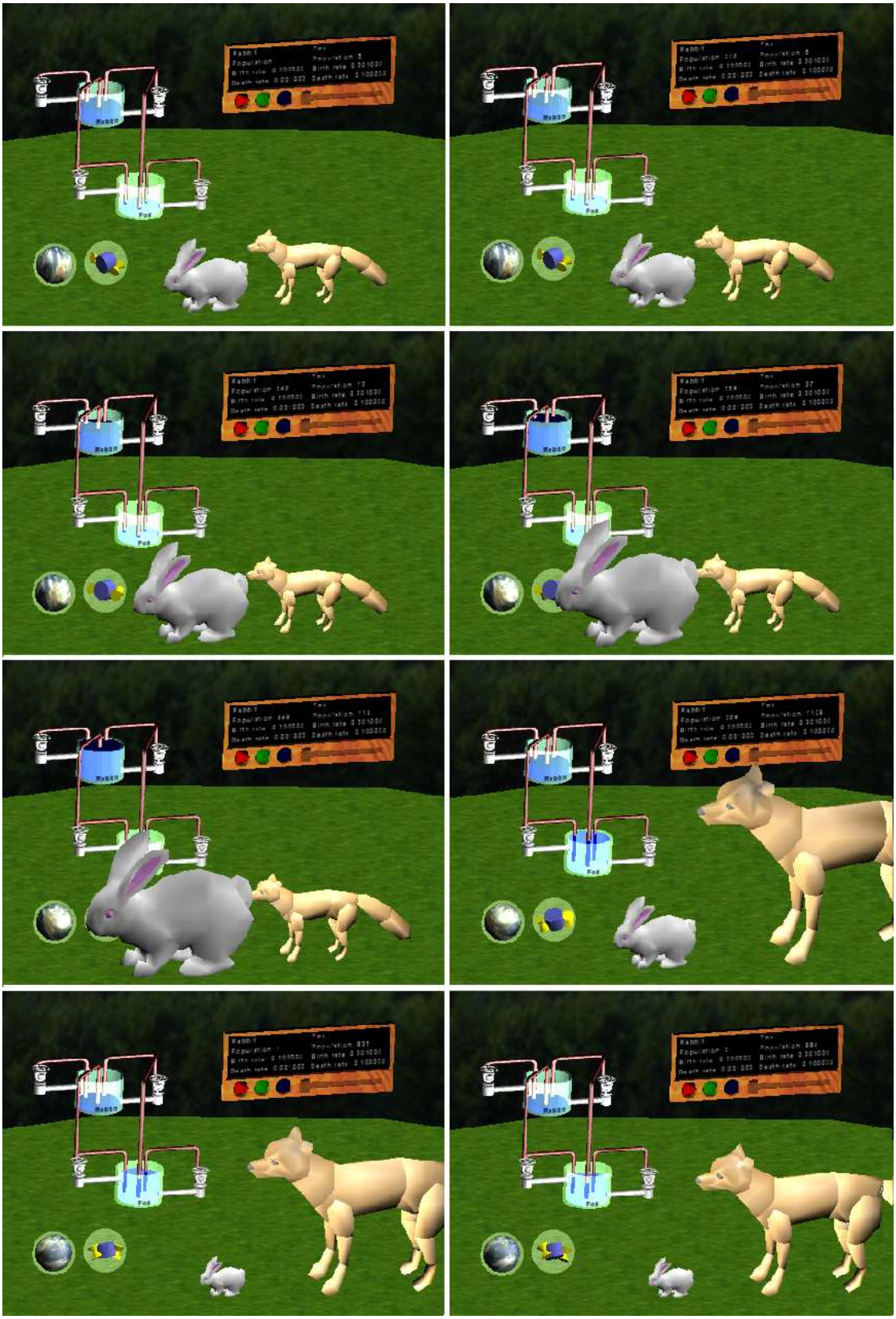

Figure 4–15. Snapshots of the integrative modeling and simulation

4.3 Discussion

This chapter introduced an integrative and interactive approach to modeling and behavior using a special type of system dynamics model, the Lotka-Volterra model. The system dynamics approach in particular provides a fluidic metaphor for representing material flows, culminating in changes in population over a period of time. A system dynamics model is generally composed of a directed network of rates, levels, auxiliary variables, and constants. System dynamics is especially useful for modeling a system with feedbacks. It starts by capturing the feedback structure of the system in a diagram called a causal graph. The causal graph is then converted into a flow graph, which is the base of the simulation model. Creating a causal graph for the Lotka-Volterra model and converting it into a flow graph are further discussed in this chapter. For model visualization, a 3D integrative modeling approach is investigated where different model types, such as a dynamics model and a geometry model, exist together within the same virtual space. The linkages between related components from different model types are represented in three dimensions so that interacting with one model's components provides context for the other model's components through a method of highlighting.

CHAPTER 5
INTERACTIVE AND INTEGRATIVE APPROACH TO VISUAL PROGRAMMING

For programming tasks that require spatial reasoning, a visual approach provides close mappings between the problem domain and the program domain, therefore enhancing problem–solving capabilities. A visual approach allows programmers to focus better on the flow of data and the spatial reasoning than sequential and textual approaches allow. Chapter 3 introduced an integrative approach to visual programming for graphics programs. With this approach a 3D output graphics is located within the same 3D virtual space where the source program exists. The framework was developed at the level of application programming interfaces (APIs) in such a way that icons represent simple operations for graphics programming, and they are not regarded as a programming languages construct.

This chapter introduces a 3D visual programming language, VPTinker, for computer graphics programming that leverages the data–flow paradigm and a factory metaphor for its conceptual model and representation. VPTinker applies a visual programming approach to learning and practicing data–flow graphics programs. The use of a factory metaphor in program visualization naturally and intuitively reveals the concepts of a data–flow paradigm and surfaces it in the program configuration. The use of a factory metaphor lends the layout of VPTinker programs to the 2.5D space where icons are laid on the floor. The limited movement of 2.5 dimensions is less confusing and provides intuitive mapping to 2D interaction devices, such as a mouse. This allows novices to easily grab and move icons using a mouse while freely navigating the 3D program space

without special training. The construction of a Tinkertoy model is represented as a data–flow network. The visual approach to data–flow programming surfaces the flow of data in the program's configuration, which makes it easy to learn and easy to practice data–flow paradigms. An intelligent learning environment has been developed for VPTinker programming and debugging.

The integrity and interactivity approach to VPTinker includes the following features:

- Programming 3D graphics via 3D language
- Integrating the output model with the source program in the same scene
- Adding interactive linkage between a source program and the output model
- Executing and debugging the 3D program within the 3D space

The following sections introduce designing VPTinker and developing an integrative and interactive environment for VPTinker.

5.1 Conceptual Language Description

VPTinker is an iconic language for programming of simple geometric object manipulation. It provides visual syntaxes and semantics for the construction of virtual Tinkertoy models. This section explains detailed requirements, design issues, and the specification of VPTinker.

5.1.1 Language Requirements

Tinkertoys are modeling construction sets similar to Lego sets. The two types of objects in Tinkertoys are spools and sticks. A spool has nine connecting holes shown in Figure 5–1. Building a Tinkertoy model is achieved by taking sticks and connecting them to one of nine holes on a spool.

To allow programmers to build virtual Tinkertoy models, a language should provide a way to specify types of objects and a way to assemble them. One way to design such a language would be using the analogy from construction of real

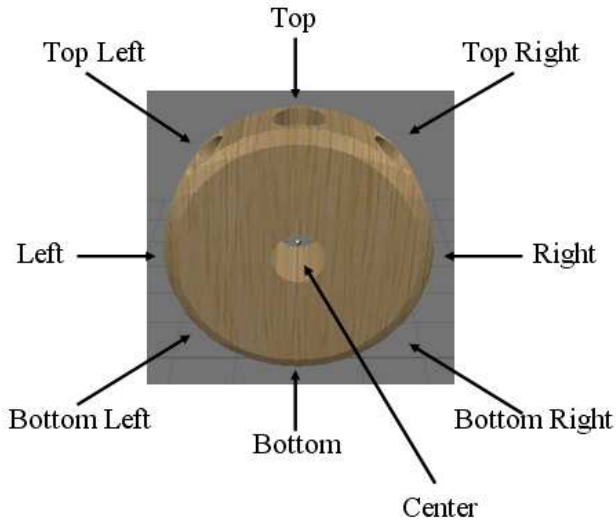

Figure 5–1. Hole positions on spool

Tinkertoy models. Before we attempt to design this language, we need to examine a use case. The use case is a scenario about what we would do to build a simple model in Figure 5–2 in the real world. If we build the model in Figure 5–2 from the top, then the order of construction would be 1) take one spool; 2) take one stick; 3) connect the stick to the bottom of the spool; 4) take another spool; and 5) connect the previous stick to the center of the new spool.

From this use case, we can define the following requirements that VPTinker should support:

- Primitive object types: spool and stick
- Connecting process or function
- Identifiers for nine connecting positions on a spool
- Group type for a connected component of one spool and one stick resulting from connecting processes

Inputs to the connecting function consist of two objects and the connecting position information. The connecting function returns a grouped component of two inputs. A group is not a physical object but a concept. In VPTinker

Figure 5–2. Simple Tinkertoy model

programming, for simplification, a group–to–group connection is not allowed. Due to this constraint, it is not possible to build a model containing loops or circular connections.

5.1.2 Computational Model

In designing the language for constructing Tinkertoy models, we followed a data–flow paradigm. In a data–flow paradigm, there are four types of elements: external entities, processes, data flows, and data storages. The external entities are the source or destination of data. Processes take data as input, process the data, and output the data. Data flows is a movement of electronic or physical data. Finally, data storages are physical external spaces that store final data.

From the data–flow perspective, we can understand spools and sticks as data and connecting behaviors as processes. Figure 5–3 shows a data flow diagram for the construction of the model in Figure 5–2. Rectangular boxes are used to represent data and processes. Arrows are used to represent the flow of data between data sources and processes. Information about connecting positions for each connect process is treated as control data. For each connecting process,

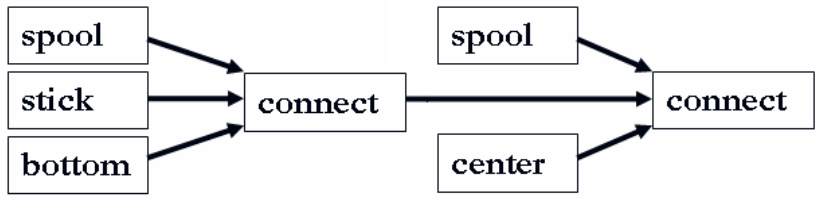

Figure 5–3. Data–flow diagram for the construction of the model in Figure 5–2

information, such as one spool, one stick, and the connecting position on the spool, must therefore be available.

5.1.3 Language Constructors

This section introduces the designing of icons in VPTinker. The basic idea of designing VPTinker icons is transforming the 2D construction flow diagram, shown in Figure 5–3, into the isomorphic 3D flow graph using metaphoric icons. With this approach, vertices and edges that appeared in a 2D flow graph will be mapped into 3D metaphoric icons. To design an iconic language for Tinkertoy construction, we need icons to represent spools, sticks, connecting positions, connect functions, and data flows. The decision for a right metaphor to represent language constructors involves more than one factor.

Metaphors are found to be beneficial in designing notations for conceptual systems. The decision for a right metaphor to represent a system involves more than one factor. The right metaphor should be the one familiar with users and should provide an insight into the concept that is represented. A factory metaphor is applied in designing icons to the extent that when the iconic visual language is taught to students, the nodes in the data–flow graph are said to *behave like* machines that take input geometry pieces, operate on them, and produce a final result. Spools and sticks are represented as materials. Connecting functions are mapped to stations where input materials are assembled by imaginary workers. A

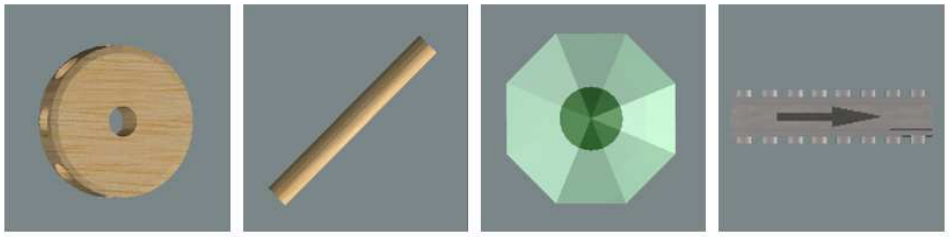

Figure 5–4. Icons for Tinkertoy model programming

data link is represented by a conveyer belt where materials move alone between stations. The icons of VPTinker are shown in Figure 5–4.

Spool and stick. Starting from the left in Figure 5–4, the first two images show the icons for the spool and the stick. They look exactly like the physical Tinkertoy construction sets. Declaring spools and sticks is done by direct manipulation, such as clicking or dragging icons within the programming environment. Details directing manipulation of icons for declarations are explained in Section 5.2.

Connect. The third image in Figure 5–4 shows the connect icon. The connect icons take exactly two inputs and assemble them to produce a grouped object. For this connect icon, there is no specific object representing the inputs and the outputs. The connect icon was able to be designed without input and output ports since VPTinker does not allow the group–to–group connection. Having no explicit objects for inputs and outputs of the connect icon would increase programming errors. This is because, the expected numbers or types of inputs and outputs are not explicitly visualized. However, having no explicit objects for inputs and outputs of the connect icon provides a flexible layout since inputs and outputs can be linked to and from any part of a connect icon. When a group type object, which is the output of a connect icon, is provided as an input for other connecting icons, we can simply add a trace from the source connect icon to the other connect icons. With this design, the types of inputs for connect icons must be referred from the

source of inputs. Since there is no group–to–group connection, at least one input to the connect icon must be a type of **spool** or **stick**, and at most one input is a type of **group**. When a **group** type object is used for an input of a connect icon, we can infer which part of the **group** type object must be used in this connection by looking at the type of another input. If another input is a type of **spool**, then the stick part of the group type input should be used for the connection. If another input is a type of **stick**, then the spool part of the group type input should be used.

Trace. The last image of Figure 5–4 shows the icons for the trace. With the data–flow approach, the flow of data is usually represented as arrows. Here, the conveyer belt introduced in Chapter 3 is used to bridge two icons and to represent the flow of data. We can also interpret the trace icon as a representation of a parameter passing to the connect icons.

Connecting points. Icons for the connecting positions are designed as a part of the connect icon, not as a separate icon. In Figure 5–5, connect icons are shown with nine different connecting position icons. Connecting positions are represented by red triangles except for the center, which is represented as a red cylinder. These connecting positions show how to connect the spool and the stick, which are linked to each connect icon.

5.1.4 Layouts and Programs

Increasing research claims 3D visualization displays more information than 2D visualization, and extensive methodologies have been introduced for effective 3D visualization for various information domains [52, 53, 93–95]. One disadvantage of 3D visualization is, without proper 3D input and output devices, manipulating and navigating are not trivial tasks.

The term “2.5D visualization” is used for describing models where 3D objects are arranged on a 2D surface. The 2.5D visualization provides dense and clear

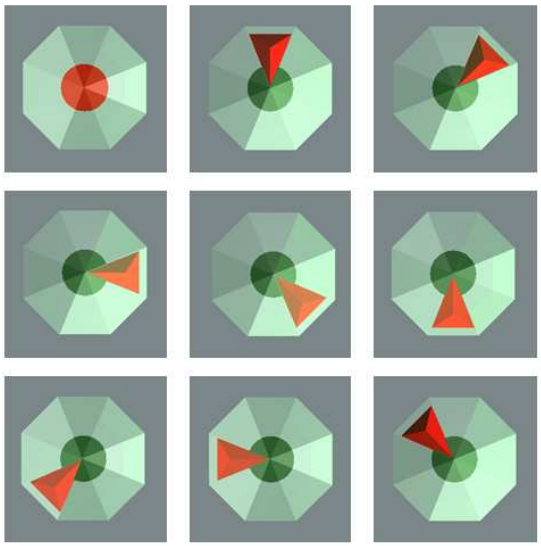

Figure 5–5. Icons for connecting positions

visualization for information [60]. The limited movement in 2.5 dimensions is less confusing and provides intuitive mapping to 2D interaction devices, such as a mouse [96]. Several research leveraged 2.5D layout in software visualizations [50, 51, 54–58]. Robertson et al. [97] and Tavanti and Lind [98] introduced their visualizations in three dimensions, however, their visualizations were actually in 2.5 dimensions. Hopkins and Fishwick [59] and Balzer et al. [60] introduced a mixed approach to the visualization in three dimensions and 2.5 dimensions where the entire layered system is visualized in three dimensions while details of each component is visualized in 2.5 dimensions. The choice of 2.5D or 3D visualization is dependent upon characteristics of the information domain and its representational metaphor.

The VPTinker program for the model in Figure 5–2 would look like Figure 5–6. The structure of this program is the same as the diagram in Figure 5–3, except the connecting icons take only two inputs. The red marks on each connecting icon describe the way of assembling the input materials. The red triangle at the bottom of the left connecting icon shows that the stick linked to this

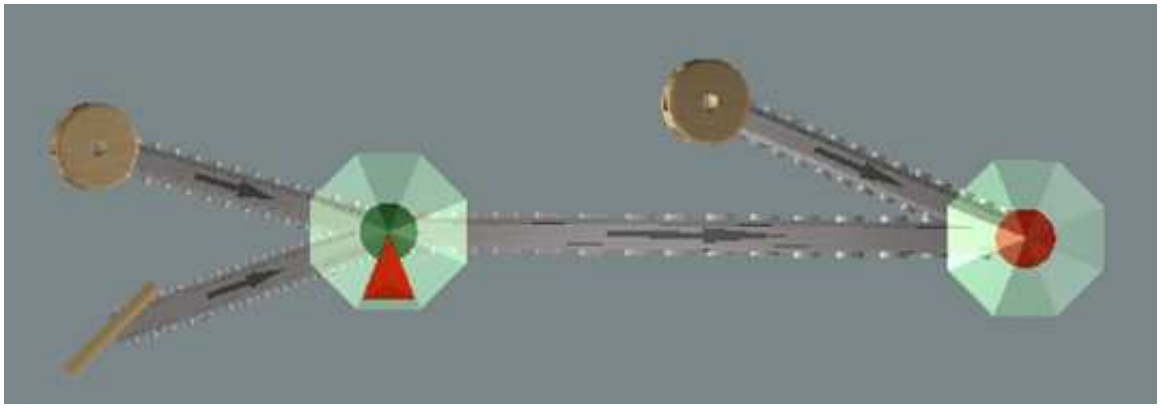

Figure 5–6. Iconic program model for the model in Figure 5–2

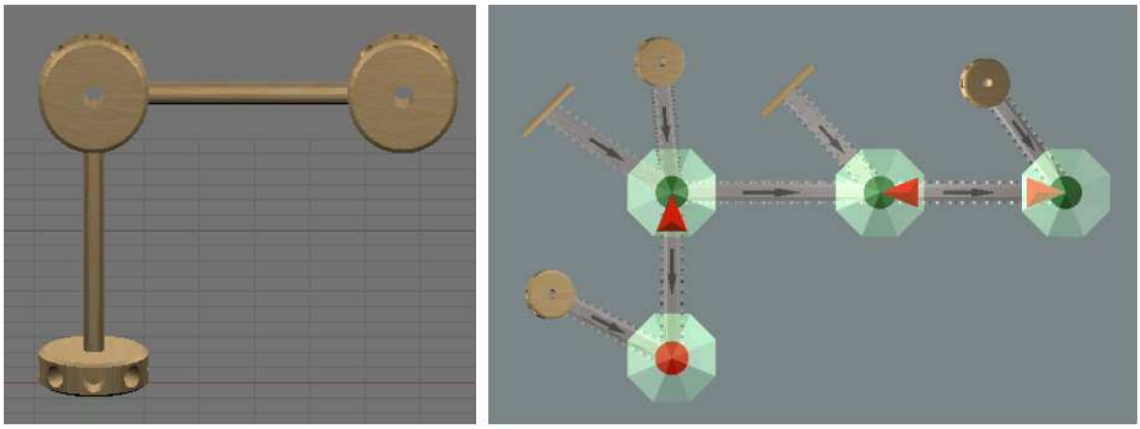

Figure 5–7. Tinkertoy model visual program

icon must be connected to the bottom of the spool, which is linked to this icon. In the same way, the red cylinder at the center of the right connecting icon shows that the stick linked to this icon must be connected to the center of the spool linked to this icon. As explained earlier, one way to understand building the Tinkertoy model is using a factory metaphor with a production line. Materials move along the conveyer belt and go through the connecting stations represented as octagons. As they go through stations, the imaginary worker assembles the materials and places the product onto the outgoing conveyer belt, if there is any. Figure 5–7 shows more complex Tinkertoy model and its iconic program.

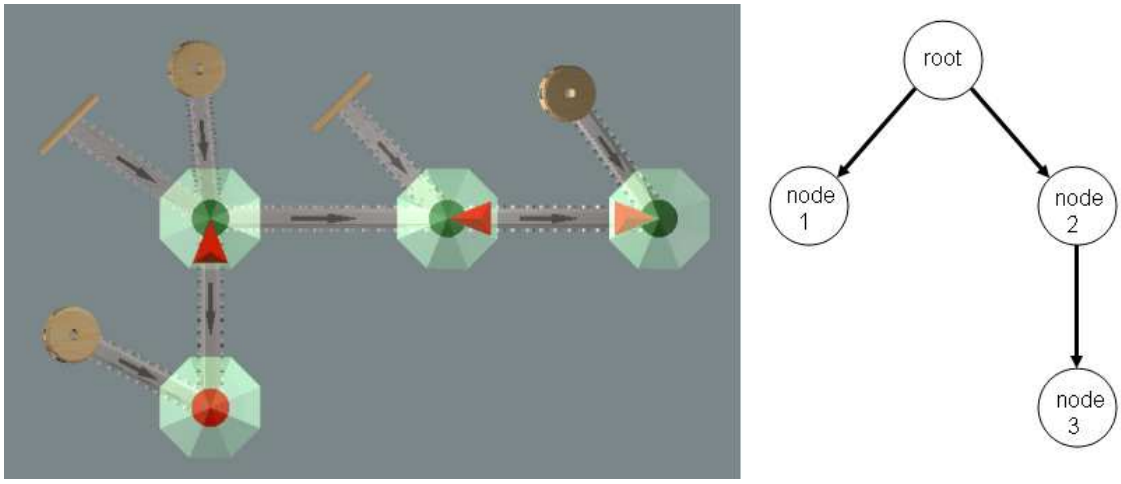

Figure 5–8. Example of visual program and internal tree representation

5.1.5 Internal Representation

Internally, VPTinker programs are stored in a tree structure. During the process of transformation from the source program into the tree structure, only connect icons and traces between connect icons are converted into nodes and edges, respectively. Spools and sticks are not translated into the tree structure, but their information, such as names, are stored together with the connect icons that they are linked to. A connect icon that does not have another connect icon as an input becomes the root of the tree. The parent and child relationship is set between neighboring connect icons. The connect icon that provides its output to another connect icon becomes a parent, and the connect icon taking the input from another connect icon becomes a child. Figure 5–8 shows the internal tree representation of the model shown in Figure 5–7. VPTinker programs are executed by the interpreter visiting each node of the internal tree structure.

5.2 VPTinker Programming

This section introduces an environment for VPTinker programming. The VPTinker environment, which is built upon GameBlender [61], allows users to specify the construction flow of Tinkertoy models using the VPTinker icons. Programming with the VPTinker environment is straightforward: Simply place the

spool, stick, and connect icons within the program space and link them properly using trace icons.

Figure 5–9 shows the programming environment for constructing Tinkertoy models. The control panel located at the upper left provides most interfaces for the creation of VPTinker programs. Figure 5–10 shows a detailed image of the control panel with brief explanations. The spool and the stick icons, which are located at the top of the control panel, are directly manipulated by dragging and dropping. To include spools and sticks within the program model, users directly grab and drag them out of the control panel. Connect icons are added in the program space not through direct manipulation but by clicking the connect button written as *CONNECT*. The direct manipulation approach is not applied to the connect icons because, unlike the icons of spools and sticks that represent physical objects in the real world, the connect icons represent the abstract action or behavior. The conceptual difference between getting physical objects and performing an action is reflected through the interface for getting the icons. The rest of the three buttons found in the middle of the control panel are for editing programs. Icons and the sliding bar at the bottom are used to start, pause, stop, and change the speed of the program execution. Details about the functionality of these interaction buttons are elucidated in Section 5.3.

The following directions explain how to use the main interface for Tinkertoy programming:

- To get a spool or a stick icon, press the left mouse button (LMB) on the icon and drag it out of the control panel.

- To get connect a connect icon, click (press and release) LMB on the button written as *CONNECT*. This will include a connect icon within the program space.

- To link icons using the trace icon, first press the right mouse button (RMB) on the source icon. This will highlight the source icon by blue. Once the

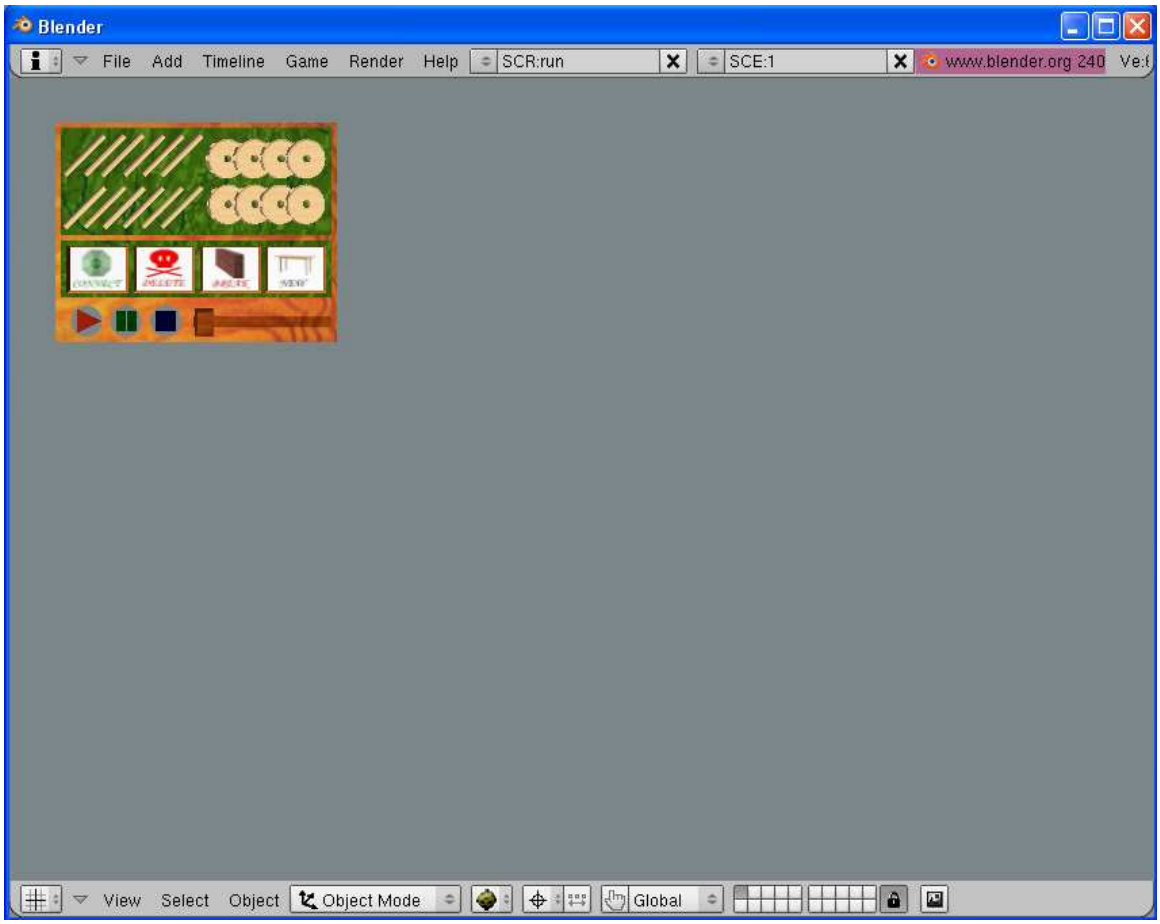

Figure 5–9. Tinkertoy programming environment

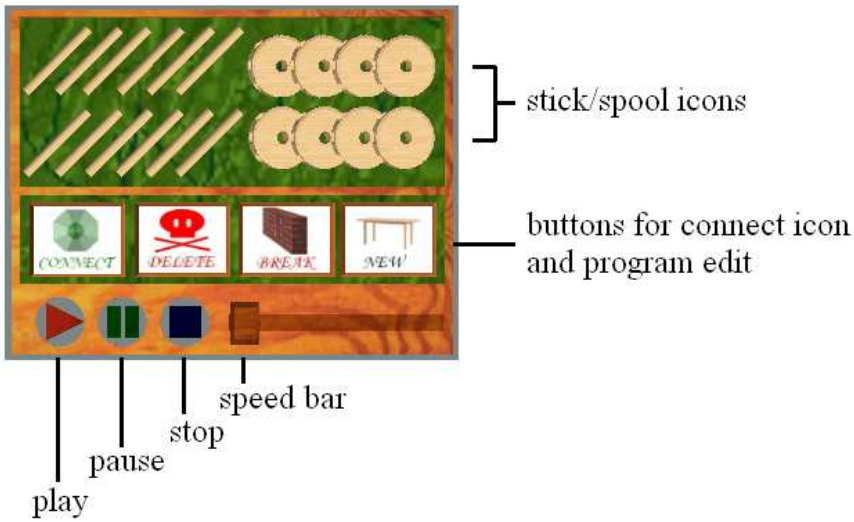

Figure 5–10. Main control panel

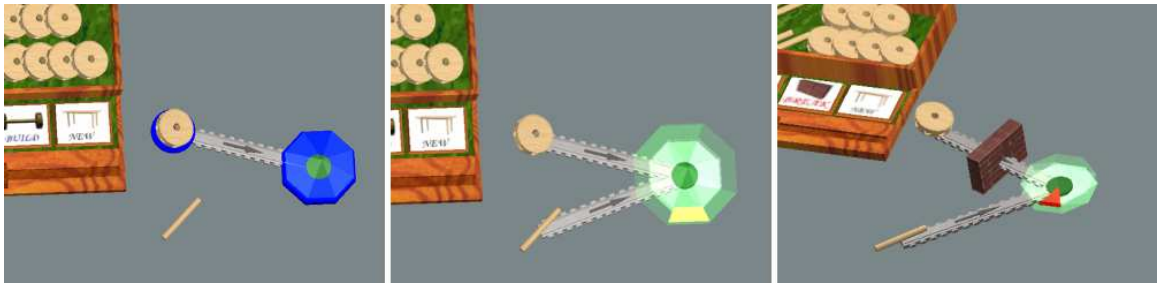

Figure 5–11. Snapshots for interface usages: From left, setting trace, setting control point, and setting break point

source is highlighted, move/drag the mouse to the target icon while pressing RMB. Once the mouse cursor is on the destination, the target icon will be highlighted by blue, and the trace icon will appear between the source and the target. Releasing RMB on the target while it is highlighted will leave the trace between two icons. The first image of Figure 5–11 shows the snapshot of setting the trace between icons.

- To set a connecting point for a connect icon, click RMB on the connect icon. This will pop an interface for setting the control points. There are a total of nine buttons on this interface, and each button is highlighted when hovering the mouse over this interface. Each of the nine buttons represents the nine connecting positions specified in Figure 5–1. The second image of Figure 5–11 shows the situation where the interface for the connect icon is activated, and the button for the bottom connection is highlighted. Clicking on this highlighted button will insert the red triangle at the bottom of the connect icon.

- To delete icons from the program space, first click on the icon using LMB. Selecting multiple icons is achieved by clicking LMB on icons while holding the *Shift* key. Once the desired icon(s) is selected, clicking the button that says *DELETE* from the main interface will delete the selected icon(s).

- To set a breakpoint, first click LMB on the trace icon where you want to halt the program execution for close examination. Once the desired trace is selected, clicking the button that says *BREAK* from the main interface will place the 3D wall for the selected trace. The last image of Figure 5–11 shows the snapshot that a breakpoint is set between the spool icon and its adjacent connect icon.

5.3 Integrative and Interactive Environment

An integrative and interactive environment has been developed for learning, programming, and debugging VPTinker programs. This environment provides

a holistic view for the program source and the execution output by integrative visualization. It also provides enhanced interactions to serve as an intelligent, comprehending, and learning tool for VPTinker programming.

5.3.1 Integrative Visualization

Multiple views is a technique that uses more than two distinct views to support the investigation of a single conceptual information or system [29]. Multiple views allow viewers to focus on partial information while they are linked. The information contained in individual views can therefore be integrated into a coherent view of the information as a whole. This section introduces an intelligent 3D learning environment for programming, learning, and debugging of VPTinker programs via interactive animation. This environment integrates the program source view and its execution output view within the same virtual space while preserving and enhancing the interactions provided with multiple views, such as linking and brushing [30–35]. A source program and execution are two distinctive perspectives of problem–solving. Each of them describes a static and dynamic perspective, respectively, and they are semantically tightly coupled. The graphics approach to graphics programming makes it possible for a 3D program to be integrated with its 3D output graphics. This section introduces an integrative approach to VPTinker programs that blends both program syntax and semantics within the same virtual space.

With traditional graphics programming approaches, the source program and the output graphics are two distinctive entities. With the integrative approach, the program source and the output graphics can be treated as coexisting entities. Within the integrative environment, programmers can watch the output graphics being built as the input program is executed and animated. As the program execution progresses, semantically related objects from the program source and the output are simultaneously highlighted. This type of visualization provides a

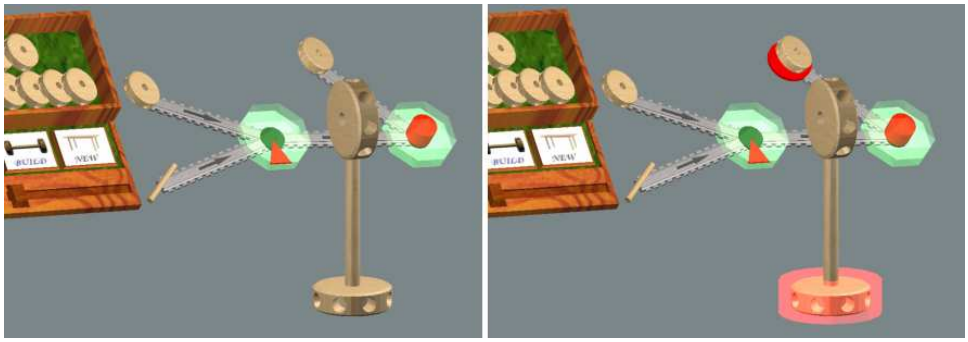

Figure 5–12. Model integration for the Tinkertoy program

holistic view of program execution. This suggested mode of operation integrates *syntax* with *semantics*, where *semantics* is loosely specified to be equivalent to *program behavior*. Learning the elements of computer graphics is facilitated with this integrative programming environment. Constructing a 3D graphics program generates 3D animated graphics, which is juxtaposed with the input program. A dual interaction exists between the integrated source program and the output. The interrelationships among related components are surfaced in the user interface in such a way that clicking on one component from the source program highlights its correspondent component from its output and vice versa. Figure 5–12 shows the integration of the source program and the execution output. The related component identification through highlighting is illustrated in the image on the right. The program icon of interest is highlighted with the red cylinder, and its corresponding object from the output graphics is highlighted by the enclosed red semi–transparent cylinder.

Users can watch the output graphics being built while the input program is executed and animated within the same virtual space. The execution of a program is leveraged by synchronous highlighting between the program execution points and the output construction point, which provides a holistic view of the program execution. The integrative approach to the VPTinker programming enhances the feeling of engagement and immersion by allowing modelers to navigate the

scene and view the 3D program and its output graphics from various viewpoint perspectives.

5.3.2 Interaction and Debugging

Much research has been conducted about developing efficient learning and comprehending methodologies in teaching concepts of computer science. Since the 1980s, software visualization and algorithm animations have been intensively studied to assist in understanding and learning for software and algorithms [99–101]. Brown [102] introduced the multi–view editing approach to algorithm animation. While the usability of animation for learning algorithms is still controversial, algorithm animation is gaining more interest as a learning tool [103–107]. Non–interactive visualizations, which let viewers ramain passive, have little advantage in understanding and learning the visualizations. Naps et al. [108] argued that visualization has an educational value when it engages learners in an active learning activity.

The integrative environment introduced in the previous section was enhanced as an intelligent environment for learning and practicing VPTinker programming. Within this intelligent learning environment, the progress of the program execution is demonstrated by visual means or animation. Programmers are allowed to interactively examine the program execution in 3D space by placing metaphoric walls among the program constructors to set breakpoints. Using the 3D breakpoints, the execution of the program is animated in such a way to allow programmers to visually trace the status of the program execution at each breakpoint. To leverage the integrative approach, a special type of debugging is provided so that the related components from the source program and the execution output are simultaneously highlighted for component identification on a programmer's demand. Interaction via component highlighting is enhanced by the play–pause–stop execution mode that lets programmers stop and resume

the execution for inspecting partly constructed outputs. This style of execution facilitates evaluation of programs by allowing users to 1) stop the execution in the middle to check work so far; 2) to find out how much progress has been made; or 3) to check their stage in the work. Combined with component identification, setting breakpoints and the play–pause–and–stop execution mode allow programmers to check a program fragment easily at any point of program execution. This type of progressive debugging environment is especially beneficial to novices. The programmer's control over the program execution is amplified via adjustable speed of program execution. Three-dimensional progressive debugging makes it practical to monitor the program execution intuitively, to do early testing and demonstration, and to make corrections sooner.

Play–pause–and–stop execution. It was mentioned earlier that user–created VPTinker programs are transformed into a tree representation. Execution for the VPTinker programs starts by creating virtual spools and sticks for each spool and stick icon from the source program. Once all spools and sticks are created, the tree representation of the source program is handed over to the interpreter for execution. The interpreter visits the root and its descendants of the tree, based on the depth first search (DFS) algorithm. Upon visiting each node, the interpreter simply connects two inputs of the current node as its connect position specifies. As mentioned earlier, only connect icons are converted into tree nodes, along with the information required to perform the connect operation: two inputs and the connecting position.

Execution of VPTinker programs is initiated by clicking the play button, which is the red triangle located at the bottom of the control panel. Users can pause the execution of the program using the pause button–the green vertical bars next to the play button–and they can closely examine the partly constructed output. The execution resumes when the play button is clicked again. Terminating

program execution is achieved at any point of execution by clicking the stop button, which is the dark blue box located next to the pause button. The sliding bar next to the stop buttons is used to control the speed of execution. These buttons and the slide bar provide the sophisticated manipulation of program execution. For example, users can stop the execution of the program at any point and change the execution speed or examine the partly constructed output.

Three–dimensional breakpoints. Breakpoints indicate a point at which the debugger halts execution of a program. They are used to examine the status of the program execution at some point and to step through the execution. Setting breakpoints is one of several common debugging mechanisms. In the VPTinker environment, breakpoints are set on the trace icons using 3D walls. Detailed instructions for setting a breakpoint on a trace are described in Section 5.2. Incorporated with component identification and the play–pause–and–stop execution mode, breakpoints allow programmers to check a program in a step–by–step manner and demonstration of a program fragment.

Once breakpoints are set for the program, users can start program execution by clicking the start icon. The program is executed until it meets the closest breakpoint and halts. Clicking the start button again resumes the execution, and this repeats until the program execution is terminated by reaching the last connect icon. Figure 5–13 shows the snapshots taken when the program execution is halted at each breakpoint. The top left figure shows the situation where the execution is halted after reaching the first spool, and the output spool object is included in the program space. The top right figure shows that the execution is halted after the first connect icon is executed. At this moment, all spools and sticks are included in the program space, and the execution shows a partly constructed model of one stick connected to one spool. The bottom left figure shows that the program execution is terminated and the resulting complete output graphics are placed next

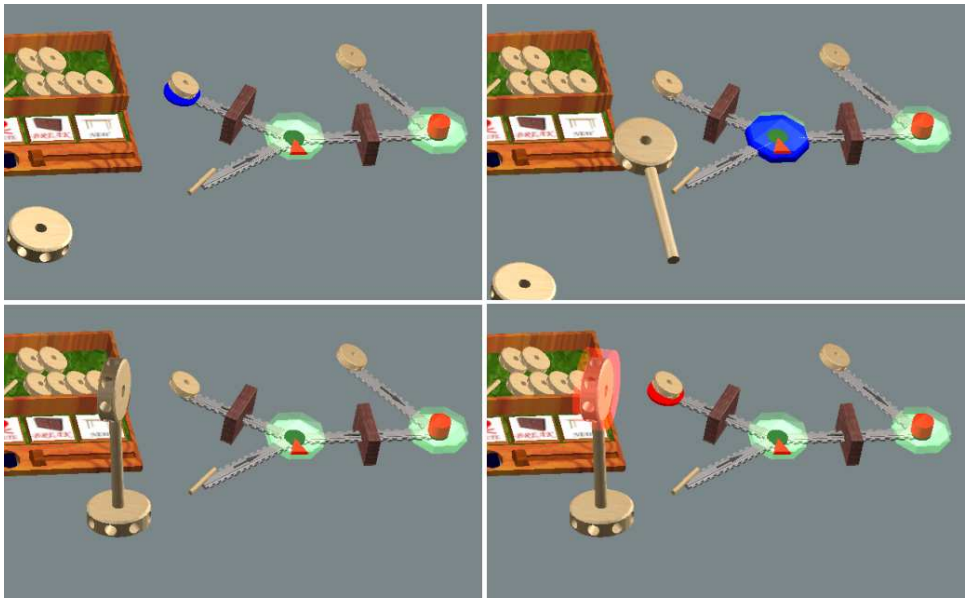

Figure 5–13. Executing Tinkertoy program with breakpoints

to the source program. The bottom right figure shows the component identification between the source program and the output model.

Error checking. Since the visual language introduced in this section is simple and users do not need to type anything, the number of errors that could rise from users' programs are limited. The interpreter for this language handles syntactic and semantic errors. The syntactic errors include 1) connecting the same types of objects; 2) having too many or too little inputs for connect; and 3) not specifying the connecting position. The semantic errors include trying to connect an object to a stick or a spool which is fully connected. When an error occurs at some connect icon, the erroneous connect icon is highlighted by a blue color, and the error message is displayed in Blender console.

5.4 Discussion

This chapter introduced a data–flow visual programming approach to geometry manipulation programming. A simple iconic language, VPTinker, has been designed for the virtual Tinkertoy model construction. A factory metaphor is applied in designing icons for program constructors. Our approach

to visual programming leverages the 3D space in such a way that both program syntax and semantics be blended within the same space. A visual approach to data–flow graphics programming provides an intelligent learning environment with educational benefits as follows:

- For programming tasks that require spatial reasoning, a visual approach provides close mappings between the problem domain and the program domain, therefore enhancing problem–solving capabilities.

- A visual approach allows modelers to focus better on the flow of data and the spatial reasoning for the model construction than sequential and textual approaches allow.

- The use of a factory metaphor in program visualization naturally and intuitively reveals the concepts of a data–flow paradigm and surfaces it in the program configuration.

- The integration of the source program and the execution output assists programmers in mapping the syntax and the semantics of programs closely via holistic visibility.

- Interrelationships among related components are surfaced in the user interface via dual interaction between the source program and the program output.

- The progressive debugging environment for 3D programs featured with setting breakpoints, the play–pause–and–stop execution mode, and the adjustable execution speed allow programmers to check a program at any point of construction.

- The proposed environment provides a holistic view for program execution via synchronous highlighting between the program execution points and the output construction point.

This chapter introduced VPTinker, which is a simple iconic language without scoping. Data types and operations available in VPTinker are limited to a small number as it was designed for novices. The limited scalability of VPTinker was also related to a practical reason. That is, we planed to investigate the problem–solving capability of VPTinker by comparing it with its textual counterpart language. Due to the time constraints in the classroom assessments, VPTinker should be

easy and simple enough to be taught during one class period, which is normally fifty minutes. Details about this investigation are discussed in Chapter 6. The scalability of VPTinker for complex programming might be improved by adding some language features, such as scopeing and control structures. Supporting primitive 3D objects and basic graphics operations would also improve the scalability in VPTinker programming. Our future research regarding the scalability of VPTinker programming is discussed at the end of Section 7.2.

Scalability is a common issue in visual programming research. To solve the scalability problem, several techniques have been introduced including nesting, fisheyeing, and zooming [109, 110]. These techniques help programmers to develop complex software using visual languages. Supporting the domain abstractions at the language level is another way of handling the scalability problem in visual programming [18–21]. As software has been getting complex, languages have been evolved in accordance with domain. Domain–specific visual languages allow programmers to build software as models of which elements represent components that are part of the domain world, not the code world.

CHAPTER 6
USABILITY TEST ON VISUAL PROGRAMMING

The goal of visual programming is to improve the problem-solving capability in programming by replacing textual notations of languages with visual ones. Visual programming languages (VPLs) may be easier to learn and understand than textual languages. However, the vision of VPLs as a general programming language is controversial, and empirical studies about the problem–solving capability of VPLs have shown conflicting results [13–17]. Unlike general purpose programming, VPLs have been successful in domain–specific programming from various areas [18–21]. Many successful domain–specific VPLs leverage a data–flow paradigm in such a way that programs are constructed as instructional pipelines or networks. We were interested in whether or not the problem–solving capability in computer graphics programming would be leveraged by a visual approach. Testing the problem–solving capability of visual programming is not trivial. One way of evaluating visual programming is to compare it with a traditional approach, which is programming via a textual language. To evaluate VPTinker introduced in Chapter 5, a textual counterpart language was designed in such a way that it was mapped one–to–one with VPTinker. Designing the textual language starts by specifying the mapping criteria between VPTinker and textual languages. This chapter introduces designing a textual counterpart language of VPTinker followed by the empirical study, comparing the problem–solving capability of VPTinker to this textual language.

6.1 Designing Textual Language

VPTinker was designed to develop a simple and complete 3D visual language for constructing virtual Tinkertoy models within an integrative and interactive

environment. One of the main interests of visual programming languages is the evaluation of their problem–solving capabilities. One way to evaluate the problem–solving capability of VPTinker is to compare it with a traditional approach, which is the problem–solving capability via a textual language.

6.1.1 Mapping between Visual and Textual Languages

Since the focus of designing the textual language is comparison it with VPTinker, designing mappings between them is a crucial issue. For an accurate comparison, they must provide the identical capabilities for problem–solving, except their notations. Everything must therefore be identical except that the syntax for one language is in visual and the syntax for the other language is in textual. For complete mapping between VPTinker and the textual language, a formalism must occur for the mapping transformation process. Four criteria of the mapping transformations have been developed: unambiguous, complete, consistent, and equivalent.

The mapping must be unambiguous. The relationship between the source language and the target language has to be one–to–one. Different elements in source program must not be mapped into the same element in the target representation and vice versa. Sometimes, additional 3D objects, which are not part of the source program, are included in the target representation to give the representation a more realistic environment [10].

The mapping must be complete. In any representation of the source, we should ensure that all elements have a counterpart in the target representation, and no single element from the source or the target is left unmapped. Mackinlay [111] referred to this completeness as expressiveness, which means a graphical sentence should exactly express the input information; that is, all the information and only the information. The rule of the complete mapping between the visual and the textual language must be strictly kept to prevent that any language feature and

element that exists in one language would not exist in the other. Every semantic and syntactic element found in the visual language must be presented in the textual language. This is important because we need to make sure, for the correct evaluation of two languages, that the only difference is the representations, so that we are only measuring the only impact of this variance. All the other controlled variables must remain constant.

The mapping must be consistent; that is, similar elements in the source program should map to similar elements in the target representation. This similarity may come from the syntax or the semantics of the source. For syntax, we must ensure that if one constructor 'x' from the input program is represented with a certain visual expression, then every instance of 'x'must be represented with that expression. For semantics, we would expect that similar semantic operations in the source would be represented by similar objects. We must therefore ensure that if two constructors have similar semantics in the source, then their counterparts in target would look similar. The consistency of mappings is related to the comprehensibility of the mappings and the target representation.

The mapping must be equivalent. Equivalence here refers to the structural equivalence between the source and target representations. The source and target representations should be mapped in such a way that relations hold among the source objects, as well as among the target objects. Gentner [112] called this type of mapping as a structural–mapping.

Figure 6–1 shows the mapping between the conceptual, the visual, and the textual languages. The solid arrow from the conceptual representation to the visual representation shows the iconic mapping from the conceptual to the visual representation. The solid arrows between the visual and the textual representations show the one–to–one mapping between them. Since the visual language is mapped unambiguously and completely to the textual language, the same mapping between

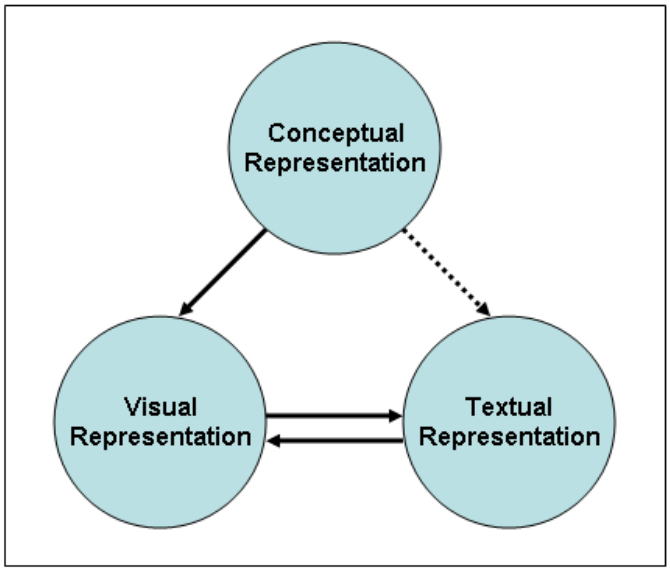

Figure 6–1. Mapping diagram between conceptual, visual, and textual languages

the conceptual to the visual can be derived from the conceptual to the textual representations. This derived mapping is shown in the dashed arrow.

6.1.2 Tinkertoy Textual Language

In this section, the textual counterpart language of VPTinker is introduced. The language follows the requirements specified in Section 5.1, and it is mapped from the visual language based on the mapping criteria discussed in the previous section. The textual counterpart language of VPTinker is an interpreted language whose interpreter is written in Python. While the textual language is designed for the purpose of the evaluation for the problem–solving capability of VPTinker over a textual one, this language itself is an independent complete language for the Tinkertoy model construction.

6.1.2.1 Language constructors and programs

Table 6–1 shows grammars for the textual Tinkertoy programming language. In this language, three types of **spool**, **stick**, and **group** exist for variables, and there is no scoping. Programs are composed of variable declarations for spools and sticks and function calls for connections. The function **connect** receives

Table 6–1. Grammars for Tinkertoy Construction Language

program → (*command* | *statement*)*
command → **stick** *ident* | **spool** *ident*
statement → **group** *ident* = **connect**(*para*)
para → *ident*, *ident*, *cont* | *ident*, *cont*, *ident* | *cont*, *ident*, *ident*
cont → **top** | **topRight** | **topLeft** | **right** | **left** | **center** | **bottom** | **bottomRight** | **bottomLeft**
ident → *letter*(*letter* | *digit*)*
letter → (**a** | **b** | ··· | **z**)*
digit → (**0** | **1** | ··· | **9**)*

three parameters: two for connecting objects and one for the connecting position information. The function **connect** returns a grouped object type of **group**.

As in VPTinker programs, either a single object or grouped objects can be passed as an input to **connect**. Three parameters in **connect** are designed to be passed in any combination to make the parameter passing mechanism of the textual language be identical to that of VPTinker. The **connect** icon has no explicit placement for the input parameters, and inputs can be connected in any order. *cont* defines nine connecting positions on a spool. *ident* stands for an identifier which is an unlimited length sequence of letters and digits, the first of which must be a letter. Identifiers must be declared before they are used.

With the language defined in Table 6–1, we can show the syntax mapping between VPTinker and the textual language, as in Figure 6–2. Using the language defined in Figure 6–1, the program for the model in Figure 5–2 is shown in Figure 6–3. The first two lines declare a spool *sp1* and a stick *st1*. The third line connects *st1* to the bottom of *sp1* and returns the connected objects *g1* as a group. At the fourth line, one more spool *sp2* is declared. Finally, in line five, *sp2* is connected to the stick of *g1*.

Figure 6–4 shows a more complex Tinkertoy model and its textual program. It is also possible to get or declare all spools and sticks before using them, as shown in Figure 6–5.

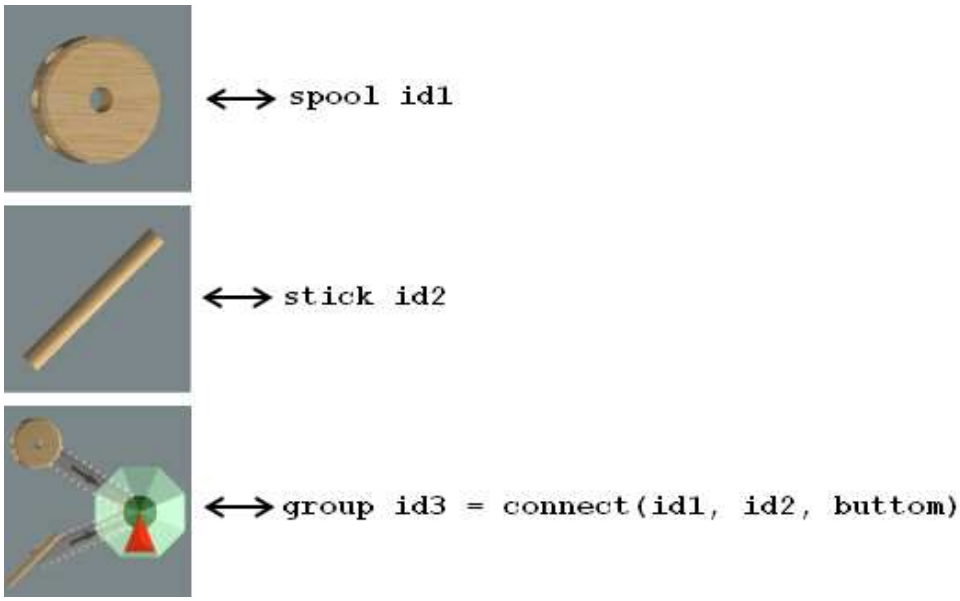

Figure 6–2. Mapping between visual and textual language constructors

```
spool sp1
stick st1
group g1 = connect(sp1,st1, bottom)
spool sp2
group g2 = connect(g1, sp2, center)
```

Figure 6–3. Textual program model for Figure 5–2

```
spool sp1
stick st1
group g1 = connect(sp1,st1, bottom)
spool sp2
group g2 = connect(g1, sp2, center)
stick st2
group g3 = connect(g1, st2, right)
spool sp3
group g4 = connect(g3, sp3, center)
```

Figure 6–4. Tinkertoy model textual program

```
spool sp1
spool sp2
spool sp3
stick st2
stick st1
group g1 = connect(sp1,st1, bottom)
group g2 = connect(g1, sp2, center)
group g3 = connect(g1, st2, right)
group g4 = connect(g3, sp3, center)
```

Figure 6–5. Tinkertoy model textual program 2

In Figure 6–5, from line 5 and 7, we can see that *g1* is provided as a parameter, along with *sp2* and *st2*. The textual language could have been designed to use *g1.stick* or *g1.spool* to specify which particular part of the group type object *g1* was to be used as an input. However, since this textual language must be strictly mapped one–to–one with VPTinker for correct comparison between them, and the **connect** icon of VPTinker does not have explicit output ports for its stick and spool, we used the group type parameter without specifying detailed information as *g1*. When a **group** type object is used as an input to a **connect** function, the interpreter can infer which specific part of the **group** type object must be used by looking at the type of the other input. In the **connect** at line 5, a stick part of *g1* is used for the connection since the type of the other parameter is **spool**. Likewise, at line 7, a spool part of *g1* is used for the connection since the type of the other parameter is **stick**.

6.1.2.2 Program execution

An interpreter, which is written in Python and run from Blender, has been developed for the textual counterpart language of VPTinker. The interpreter reads the textual programs line by line and produces 3D Tinkertoy models within the Blender 3D space. The execution results of the textual programs are identical to their corresponding VPTinker programs except that the source program is

not integrated with the execution output. The textual language interpreter is not only able to execute error–free programs but also provide features to assist debugging errors. The interpreter checks errors for syntax and semantics. Syntax error checking examines the program against the grammar specified in Table 6–1. Semantic error checking includes all the error checking handled by the visual interpreter plus undeclared or duplicate variables.

6.2 Empirical Study

Classroom assessments have been performed to investigate the impact of the 3D iconic programming on the problem–solving capabilities for novice programmers. For this investigation, the problem–solving capability of VPTinker is compared to its counterpart textual language.

6.2.1 Hypothesis and Assumptions

In this experiment, the following hypothesis and two assumptions were made.

Hypothesis:

- *Subjects in the test group show better problem–solving capabilities for geometry manipulation programming.*

Assumptions:

1. *Subjects have little or no prior knowledge of programming.*
2. *Subjects in the test group and the control group are at the same level of programming skill.*

The problem–solving capability is measured by subjects' grades on programming tasks. Our hypothesis can therefore be proven by showing that the mean grade of the test group is greater than that of the control group. The first assumption is assured by conducting this experiment on students in CIS 3022 and CIS 3023, which are the first and second entry levels of programming courses provided by the Computer and Information Science and Engineering Department at the University of Florida.

6.2.2 Experimental Design

Subjects. Sixty–nine subjects were recruited in this study and they worked individually. The subjects were recruited voluntarily among students at the University of Florida who enrolled in CIS 3022 and CIS 3023 for the Spring and Summer 2006 semesters.

The purpose of CIS 3022 (Programming for CIS Major 1) is for students–with little or no background in computer science–to be introduced to the technical aspects of the computer science field. Among other topics, this course focuses on how to solve problems, transform the solution into a program written in the Java programming language and vice versa. Topics covered in this course include the fundamentals of programming, such as variables, primitive types, operators, branches, loops, and basic object–oriented concepts. The CIS 3023 (Programming for CIS Major 2) is built upon the foundation developed in CIS 3022. In this course, students learn more about the technical aspects of the field of computer science, including additional object–oriented concepts, problem–solving, user interfacing, and system testing. As with CIS 3022, students in this course are taught problem–solving and programming in Java.

Procedure. The experiment followed the two–group, post–test–only, randomized experimental design. The random assignment of subjects assures that both groups are probabilistically equivalent in their programming skills. The post–test–only approach measures the value of the dependent variable without a pre–test. The pre–test/post–test approach, in contrast, administers a pre–test and a post–test before and after the experiment, respectively. The pre– and post–test are used to assess the amount of change on the value of the dependent variable. The post–test–only approach is used for situations where a pre–test is impossible because the participants have already been exposed to the treatment, or it would be too expensive or too time–consuming. We adopted the post–test–only approach

due to the time constraint. Details about the course and the experiment design will be discussed in the following sections.

Subjects from the test group learned VPTinker and used it for programming tasks. Subjects from the control group learned and used the VPTinker textual counterpart language. No pre–test was necessary since random assignment assured that groups were equivalent. With this design, we were most interested in determining whether or not the two groups were different after treatments or programs. We typically measure the groups on one or more measures and compared them by testing for the differences between the means. Differences between groups on post–test indicated a treatment effect.

Before the experiment started, content forms were provided to inform subjects about the procedure of the experiment and the rights they have. The experiment began with a 10–minute video tutorial on programming for Tinkertoy models using VPTinker or a textural language for the control group or the test group, respectively. After the tutorial, 10 minutes of interactive practice followed. During the practice time, subjects were asked to perform programming using the language that they learned during the tutorial. After the practice section, subjects were asked to build programs that constructed 3D virtual Tinkertoy models using the language that they had learned and practiced. The test materials are introduced in the following section. Fifty minutes were allotted for this testing. Once subjects finished the programming, they were asked to complete a survey about their experiment.

Training. Subjects from both groups had a video tutorial and interactive practice to learn and become familiar with each language. The first half of each video tutorial for both groups was identical in introducing the concepts of Tinkertoy programming. The second half introduced VPTinker and its textual counterpart language for the test and the control group, respectively. Following

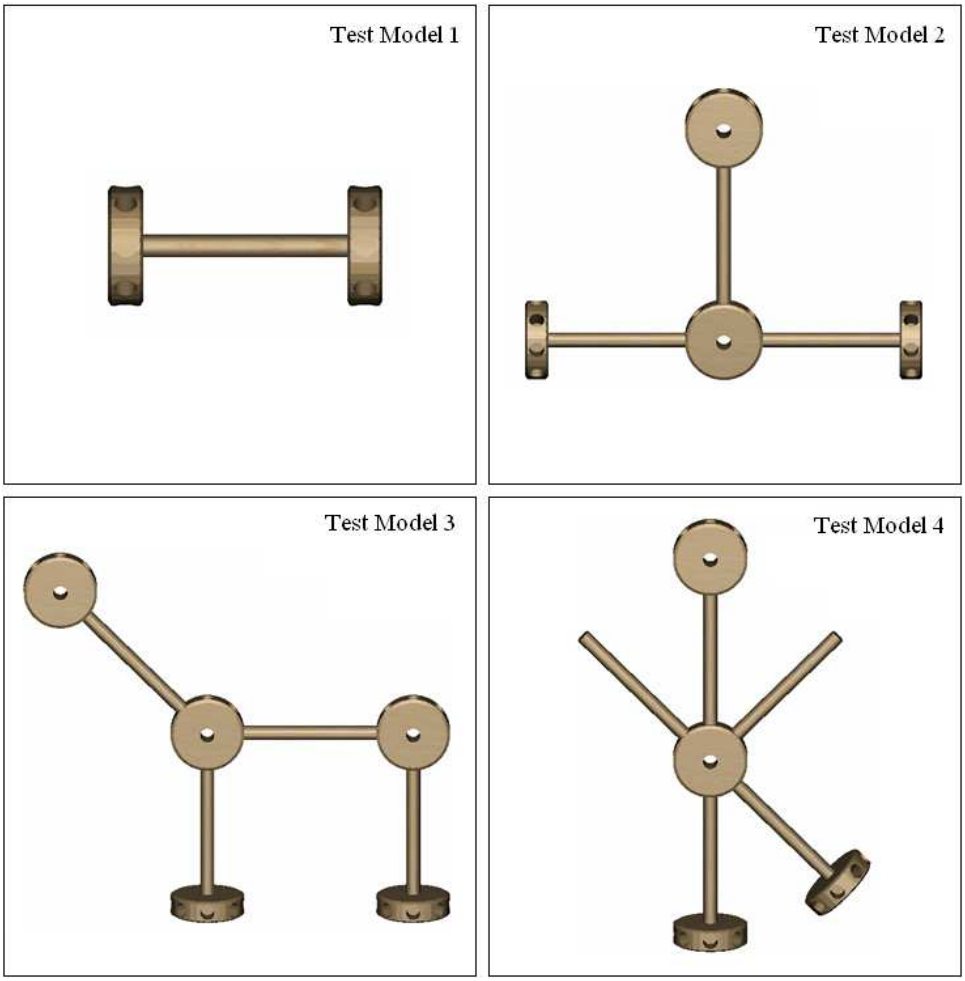

Figure 6–6. Test materials

the tutorial, subjects from both groups practiced the usage of each language for approximately 10 minutes.

Test materials. The test materials were identical for both groups and they are illustrated in Figure 6–6. Subjects were asked to program these models from the simplest one to the most complex one: Test Model 1 through 4. Appendix A contains details about the programs for these models in visual and textual languages.

Grading. Subjects' programs were graded by running them in Blender. For each task model shown in Figure 6–6, its program was regarded as correct if the program produced the same model. Subjects got one point for each correct

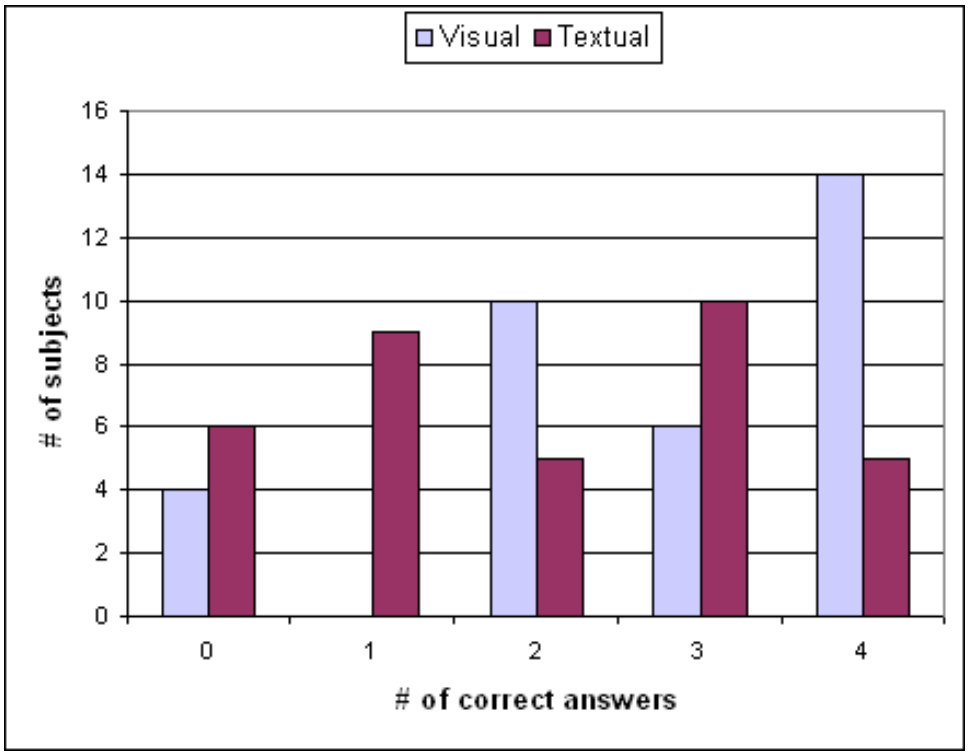

Figure 6–7. Graphs for the scores of subjects

program. Thus, the highest score a subject could get was 4, and the lowest one was 0.

6.2.3 Experimental Results

6.2.3.1 Test result

The score graph of subjects from each group is illustrated in Figure 6–7. The *Visual* and *Textual* graphs show the scores of subjects from the test and the control group, respectively.

In analyzing the experimental data, a t–test was performed with the subjects' grade from each group. A t–test is used to determine if the scores of two groups differ on a single variable. Table 6–2 shows details of a t–test analysis for our experimental data. The mean grade of the test group is higher than that of the control group. This difference is considered statistically significant based on the t–test analysis (two–tailed t–test, t=2.45, df=67, $p \leq .02$). This result supports our

Table 6–2. T–test: Two–sample Assuming Unequal Variances

	Test Group	Control Group
Mean	2.764705882	1.971428571
Variance	1.76114082	1.85210084
Standard deviation	1.327079809	1.360919116
Observations	34	35
df	67	
t Stat	2.45143115	
P(T $\leq$ t) two–tail	0.016843293	
t Critical two–tail	1.996008331	

*t: the t statistic
*df: the degree of freedom
*P: the probability that the t statistic calculated for our data is lower than or equal to the critical t–value.

hypothesis that subjects in the test group show better problem–solving capabilities provided by better grades than the control group.

As a reason for the better grades of the test group, we presumed that the VPTinker programming might motivate subjects in understanding and solving the test materials better than its textual counterpart programming might motivate. Another reason would be related to the main interaction devices. The main interaction device for the test and the control group was a mouse and a keyboard, respectively. For the programming of the same task, the minimum mouse clicks required in VPTinker programming was less than the minimum key strokes required in its textual counterpart programming. Hence, there could be more chances for the control group to make syntactic errors. Section 6.2.3.2 lists responses about VPTinker and its textual counterpart programming from the test and the control group, respectively. Section 6.2.4 discusses details about the notational usability of VPTinker and its textual counterpart language.

Table 6–3. Responses for the Multiple–choice Questions on the Survey

Survey Questions	Mean1	Mean2
What year are you in?	2.1	2.2
Was the programming easy?	4.3	4.3
Was the programming tool/language easy to use?	4.3	4.1
Did you enjoy programming with this tool?	4.1	4.1
Did unfamiliarity with the tool, if there was any, hinder your programming?	2.4	1.3
If you experienced textual programming, was programming using 3D icons easier than textual programming?	3.8	–
Do you think this new style of programming would motivate you for learning programming?	3.7	–

∗ Mean1: mean of the test group
∗ Mean2: mean of the control group
∗ The last two questions were only asked to the test group

One observation was that there existed subjects who failed all four tests from both groups. In Figure 6–7, four from the test group and six from the control group failed all four tests. From this observation, we can say that, for the certain people, the visual approach does not help problem–solving at all. That is, visual programming is not a fantasia.

6.2.3.2 Survey

The survey included multiple–choice questions listed in Table 6–3. the first five questions were commonly presented to both groups, and the last two questions were presented to only the test group. All answers were scaled from 1 to 5: 1 being "strongly no" and 5 being "strongly yes." But there was an exception: For the question of *What year are you in?*, numbers 1 to 5 were used to represent the university year of subjects. Table 6–3 also lists the mean of each survey question from the test and the control group.

The surveys also included two open–ended questions. The first question was presented only to the test group, and the second question was presented to both:

- What are the points that need be considered to improve this tool further?

Table 6–4. Responses for the VPTinker Programming

	Category	Number of Responses
Comment	Easy and simple	7
	Enjoy	5
	Useful/great learning tool	4
	Interesting	3
Suggestion	Applying to other programming	5
	Displaying output progressively	4
	Confused	4
	Space/layout management	3
	Labeling icons	1

Table 6–5. Responses for the Textual Programming

Category	Number of Responses
Confused	8
Enjoy	8
Easy and simple	6

- Any comment?

We used a standard technique in the social sciences called "coding" [113, 114] to obtain quantitative results from qualitative data. First, the responses were read and a list of possible categories was made. Next, the set of categories was edited, merging duplicate categories and sometimes splitting categories. Finally, the responses were read again, and each response was coded into one or more categories, according to the category list. Table 6–4 and 6–5 summarize the responses to the open–ended question(s) from the test and the control group, respectively.

Some subjects from the control group commented that the language was confusing:

- *There can be some confusion as to what the groups we are creating refer to.*
- *It was very unclear the order to build the groups so as to connect everything properly.*

- *The programming was not very difficult to use, but you need to be extremely careful with how you position the spools and sticks.*

Some subjects from the test group showed interest and enthusiasm:

- *Interesting way of programming*

- *This is a very interesting set up. While graphically oriented, it is based more on process than product, as a result, the program tends to look nothing like the final construct, as one would expect from most graphical programs, but instead resembles an assembly line floor plan.*

Some subjects commented about VPTinker as a learning tool:

- *This is a great learning tool*

- *I am already motivated to learn programming*

- *This programming tool may be helpful to people who have trouble with abstract thinking, but it would take time to learn.*

- *Other programming languages like Java, C, etc. make us remember a number of codes. It is easier to build a program by using 3D icons.*

Two subjects from the test group gave negative comments about visual programming:

- *I am biased towards text. Nothing visual would appeal to me.*

- *There is no challenge in iconic programming.*

For the question regarding the improvement of VPTinker programming, comments regarding the application of VPTinker were provided:

- *That would be good if this language apply to solve problem.*

- *I want more objects other than spool and stick.*

- *I doubt its application to big projects.*

Some subjects also mentioned the spatial and layout restrictions:

- *The layouts of programs can get very cluttered and confusing if they are large.*

- *Space constraints–as the program becomes more complex, the visual programming tool may have difficulty showing a large clear view of program.*

The purpose of our experiment was to evaluate the VPTinker programming language, not its programming environment. Hence, in this experiment, the progressive evaluation of programs was not permitted. Subjects were only allowed to see their program outputs once they finished their programming. Interestingly, as a way to improve VPTinker programming, some subjects suggested a feature for the progressive evaluations:

- *A preview window which you can see your progress along the way.*
- *See the process as you work on it. Meaning see the results as you built it.*

6.2.4 Empirical Study Discussion

Cognitive Dimensions of Notations (CDs) are features of computer language as information structures or notations that describe the usability of that language. The CDs' framework has been designed to assist the decision–making about notations or representations where usability tradeoffs are a design factor [115–117]. The CDs' framework has been applied in analyzing our empirical results to reveal which features of VPTinker might be creditable for the better grade of the test group. Based on the CDs' framework, the usability of VPTinker and its textual counterpart language is analyzed as follows:

- **Unambiguous abstraction gradient**: Since the textual counterpart language of VPTinker is designed to be mapped one–to–one with VPTinker, the level of abstraction is equivalent in both languages.

- **Closeness of mapping**: By means of its expressive freedom, a properly designed visual language provides close mappings or representations to domains, especially for spatial problem domains. Leveraged by freedom of layouts, VPTinker provides closer mapping to the Tinkertoy model construction than the textual language provides.

- **Consistency**: Because VPTinker and its textual counterpart have such a small set of language constructors and grammar, consistency of language does not lend itself as a significant language feature.

- **Diffuseness**: Diffuseness of programming sometimes makes the programmers lose themselves physically or mentally within the program space. Normally, visual notations require more space than textual notations, and an effective use of space is one of the huge challenges in visual programming. During the experiment, we observed that subjects from the test group visually dragged their programs to get extra space for complex programming tasks. On the contrary, none of subjects from the control group scrolled the editor for the extra space.

- **Error–proneness**: Properly designed visual languages are expected to be less error–prone semantically and syntactically. They provide close mapping between the problem domain and program domain, and manipulating visual notations would decrease errors due to typos. In fact, the better grade of the test group came from less semantic errors in their programs. An interesting finding was that the number of syntactic error types found in each group is significantly different. While the only syntactic error found from the test group was an unmatched number of parameters, error types from the control group included typos in keywords, undeclared variables, duplicate variables, unmatched number of parameters, invalid variable names, and using '.' for the parameter separator where ',' was expected.

- **Hard mental operations**: An interesting finding was that only two subjects from the test group made notes on the sheets illustrating test models to assist their programming while 19 from the control group made notes. One way to interpret this phenomenon is that translating the spatial information of the subjects' conceptual model into the textual form takes more work than translating the spatial information of the subjects' conceptual model into the visual form with VPTinker.

- **Hidden dependencies**: Using visual languages, it is possible to explicitly reveal dependencies between program components. For VPTinker programs, conveyer belts represent linkages, such as function calls and parameter passing to functions, between notations. Hidden dependencies are therefore decreased with the VPTinker programming.

- **Premature commitment**: VPTinker has less commitment to the order of creating a code than the textual language has. Unlike textual programming, which is normally achieved in linear writing, VPTinker allows programmers to create programs in any order. Programmers can even prepare all necessary icons beforehand, then they can start to build a program by wiring the icons one by one with no particular order. Initially, however, misplaced icons must be displaced to adjust the layout, which introduces an extra effort.

- **Progressive evaluation**: Although a progressive evaluation environment for VPTinker programs was developed, this environment has not been tested in

our experiment. The reason was that the purpose of the experiment was to evaluate the problem–solving capability of the language itself rather than its environment. However, evaluating the effectiveness of progressive evaluation with the integrative and interactive environment for VPTinker would be an interesting topic for future experimentation.

- **Secondary notation and escape from formalism**: While VPTinker was not designed to provide any particular secondary notation, an inherent nature of secondary notation exists for this language; that is, a VPTinker program can be laid out in such a way that the layout of the program reflects the configuration of the output model. If the connecting position of the **connect** icon is *top*, then programmers can place one input to the top of the **connect** icon. In fact, most of the subjects from the test group laid out their programs in this fashion. For the control group, limited but some secondary notations were found, including grouping the same types of statements together and using white space between these groups.

- **Role–expressiveness**: With visual notations, it is also possible to visualize their roles in their representations. In VPTinker, the icons for nine connecting positions are designed to represent their semantics visually. However, for the current version of VPTinker, it is not possible to give meaningful identifiers to icons, and this makes them less role–expressive than VPTinker counterpart textual language, which allows programmers to give meaningful identifiers for variables.

- **Viscosity**: Viscosity of VPTinker is higher than textual language, meaning that modifying VPTinker programs requires more effort than modifying textual programs.

- **Visibility**: VPTinker has poorer visibility than the textual language, as discussed in *Diffuseness* since VPTinker requires more space than the textual language requires. The integrative approach of the programming environment provides a special type of visibility for VPTinker, which is the increased visibility between the source program and the program output as they coexist within the same physical space. However, since the experiment was not about its environment, we will not consider this as a significant feature of VPTinker at this time.

Pandey and Burnett [16] presented empirical results that matrix manipulation programs can be more easily constructed in the visual language Formd3 than in the textual languages Pascal and OSU–APL. Navarro and Cañas [45] showed that the visual programming approach with Excel programming allowed for quicker

construction of a mental representation based on data–flow relationships of a program than procedural languages, C. Green and Navarro [44] showed that mental representations of the same programming task have different structures in different programming languages. Green and Petre [14] claimed that the effectiveness of notation is task–dependent. In our empirical study, we showed that visual notations would fit more closely to the mental representations than textual ones for the tasks that require spatial reasoning. We also showed that visual programming would improve problem–solving in graphics programming more than textual programming.

CHAPTER 7
CONCLUSION

7.1 Summary

This dissertation introduced a flexible visualization approach and an integrative and interactive environment for 3D programs. With the flexible visualization approach, users can lend themselves to the creative activity in designing their own icons for the basic building blocks of programs, and wire them together to construct a program. The integrative approach to visualization blends interrelated systems within the same virtual space. This work is viewed as providing a user–customized method of programming, while leveraging the ability to blend both program syntax and semantics within the same virtual space. The integrative approach has been broadened to simulation modeling, especially for system dynamics modeling. A new method for modeling the particular type of a predator–prey system is demonstrated by leveraging the power of human interaction, as applied to two different models: a system dynamics model and the population geometry model. Both models are represented in three dimensions and are fully integrated so that interacting with one model's components provides context for the other model's components through a method of highlighting. With the integrative modeling approach, the linkages between the related objects from different model types are visualized by effective interaction means.

As a way to leverage the integrative and interactive environment for visual programming, a simple visual programming language for constructing 3D graphics, VPTinker, was developed. VPTinker is a data–flow visual programming language for novices to construct virtual Tinkertoy models. VPTinker applies a visual programming approach to learning and practicing data–flow graphics programs.

A factory metaphor was applied in designing icons for program constructors. Spools and sticks are represented as materials. Connecting functions are mapped to stations where input materials are assembled by imagery workers. A data link is represented by a conveyer belt where materials move along to stations. VPTinker programming is achieved by defining an instructional pipeline using icons. An integrative environment for programming and debugging VPTinker programs was developed. This environment blends both program syntax and semantics within the same virtual space, and enhances the interaction among the source programs, the program execution outputs, and the programmer. This approach to graphics programming enables a programmer to have a holistic view on the program syntax–the source program and the semantics–and the program output. In the VPTinker environment, a program is arranged within the 2.5D plane while the program output is constructed in the 3D space. New interaction mechanisms among the source program, the output model, and the programmer has also been designed. The user interactions include simultaneous highlighting of related components between the source program and the program output; progressive animation for program execution and its output construction; the play–pause–and–stop execution modes; and adjustable speed of the execution and the animation. The VPTinker environment also includes a novel debugging feature, such as setting 3D breakpoints among the program components for the step–by–step execution.

Classroom assessments were performed in the Spring and Summer 2006 semesters at the University of Florida to evaluate the effect of visual programming on the problem–solving capability for graphics programming. The problem–solving capability of visual programming for the geometry manipulation programming was investigated by comparing the visual programming to its textual counterpart programming. In this investigation, a textual language was designed to be

mapped to VPTinker unambiguously, completely, consistently, and equivalently. The mapping formalisms ensure that experimental differences were not due to inherent language differences. An empirical study compared the problem–solving capability of the visual language to that of the textual language. In the experiment, one group used VPTinker and the other group used the textual language. Empirical results showed that the VPTinker users received better grades than the textual language users received. The textual language users responded that the programming was simple, easy, and enjoyable. In addition to these responses, the VPTinker users commented that the programming was interesting, and VPTinker would be useful in learning programming.

The primary innovation of this research is as follows:

- A customized approach to programming enables novices to learn programming through a high degree of creativity. We express this creativity in our programs through the use of personalized 3D icons [22].

- A set of "off the shelf" dynamic model types, traditionally associated with the simulation community, is leveraged in programming without necessarily attempting to institute a standard, for example, data–flow graphs, event graphs, and Petri nets [72].

- An integrative environment provides a holistic view for both the program syntax and semantic behavior by blending them within the same virtual 3D space.

- The aesthetic range of model visualization is expanded in which the aesthetics of programming and modeling are defined to include both subjective visual appreciation, as well as objective optimality criteria [67].

- For programming tasks that require spatial reasoning, a visual approach provides close mappings between the problem domain and the program domain, therefore enhancing problem–solving capabilities.

- A visual approach allows programmers to focus better on the flow of data and the spatial reasoning than are possible for sequential and textual approaches.

- The use of a factory metaphor in program visualization naturally and intuitively reveals the concepts of a data–flow paradigm and surfaces it in the program configuration.

- The integration of the source program and the execution output assists programmers to map the syntax and the semantics of the program closely via holistic visibility.

- Interrelationships among related components are surfaced in the user interface via the dual interaction between the source program and the program output.

- The progressive debugging environment for 3D programs featured with setting breakpoints, the play–pause–and–stop execution mode, and the adjustable execution speed, allowed programmers to check the status of the program execution at any point.

7.2 Further Research

In this dissertation, we investigated the usability of VPTinker by comparing it to its textual counterpart language. The VPTinker environment, however, is introduced without its usability evaluation. Our future research includes an empirical study evaluating the impact of the proposed integrative environment for the 3D visual programming and debugging. The investigation of this environment can be further divided. First, the effect of the progressive evaluation for 3D programs within the integrative environment can be evaluated. The three features are included in the VPTinker environment for the progressive evaluation: 3D breakpoints, the play–pause–stop execution mode, and the progressive animation for the execution output. These features allow users to stop in the middle to 1) check their work; 2) find out how much progress has been made; or 3) check the stage of their work. Second, the role and effect of the dual interaction for understanding programs within the integrative environment requires empirical evaluation.

As a future research project, we can further enhance the integrative environment as a learning tool. One way to achieve this goal is adding a gaming feature. Gee [118] identified several methods that could and should be applied to get

people to enjoy learning from what the designers have achieved in attracting game players. Naps et al. [108] argued that visualization has an educational value when it engages learners in an active learning activity, and, unlike learning, gaming makes learners feel like active agents, not just passive recipients. As a way to leveraging gaming within the proposed environment, provision of instructions for VPTinker programming can be customized to fit an individual's learning styles. The instructions can be provided either as a narrative or as a "*On Demand* and *Just in Time*" tutorial upon a user's need. With the gaming approach, it is also possible to design learning and problem–solving in programming in several stages or levels. Especially, early and simple programming tasks can be designed to lead programmers to form good guesses about the solution of harder programming later on. With this approach, programmers can choose a stage that fits their programming skills, and then move to higher or lower stages as necessary. This style of learning programming allows programmers to check their understanding of programming and offer feedback in such a way that different learners feel programming is challenging but doable.

We can expand the application of the integrative visual programming approach to more general graphics programming by providing 3D primitive graphics objects, such as boxes, spheres, and cylinders. We can also provide basic operations, such as transformations, textures, materials, extrusions, delete, merge, and reading geometry data from a file. Houdini [119] introduced a visual approach to 3D graphics called channel operators (CHOPs) to provide a framework for managing interactive work and controlling the 3D elements of Houdini in real time. Using Houdini, users can construct a 3D scene directly from the 3D window while the history of the scene manipulation made by the user is recorded as an operational network. Users can also directly manipulate the operational network to trace the scene manipulation to make modifications on what they have done previously.

Houdini's CHOPs provide a non–linear and non–destructive environment for managing and programming the process of the 3D scene construction. We can adopt the CHOPs mechanism introduced in Houdini for general graphics programming within the integrative approach by facilitating primitive objects and basic graphics operations in 3D graphics construction. Once primitive objects and basic graphics operations are supported, the integrative environment can be used as a general graphics programming tool that allows both a holistic view and novel interactions.

APPENDIX A
EMPIRICAL STUDY MATERIALS

Appendix A shows experimental test materials and their visual and textual programs. For visual programs, the integrative and interactive resulting models are shown together with the source programs.

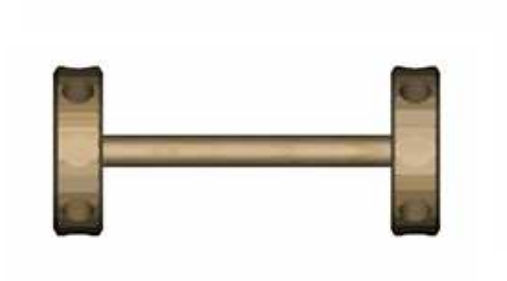

Figure A–1. Test model 1

```
spool sp1
spool sp2
stick st1

group g1 = connect (sp1, st1, center)
group g2 = connect (g1, sp2, center)
```

Figure A–2. Textual program for test model 1

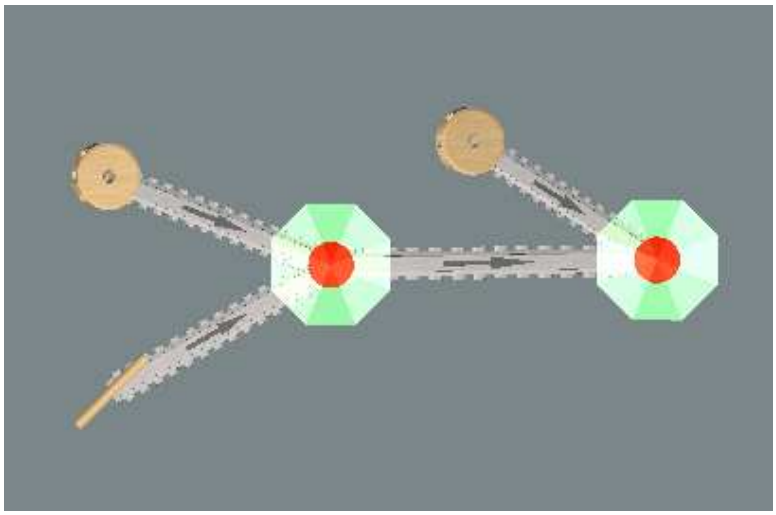

Figure A–3. Visual program for test model 1

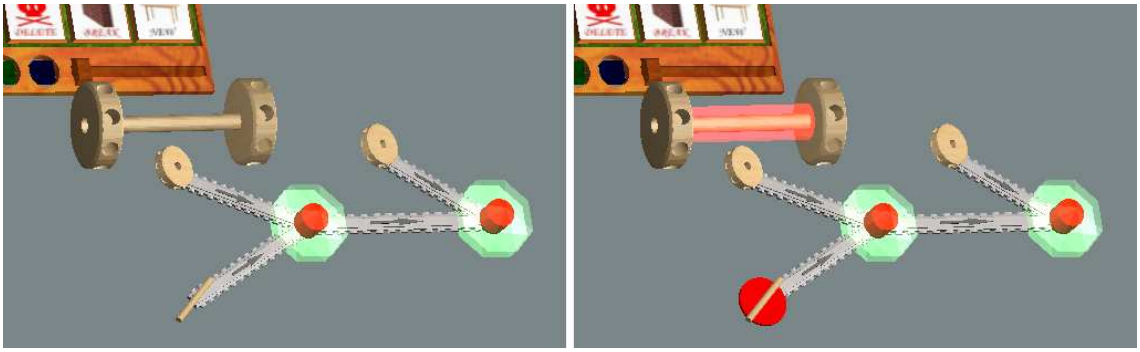

Figure A–4. Integrated model for test model 1

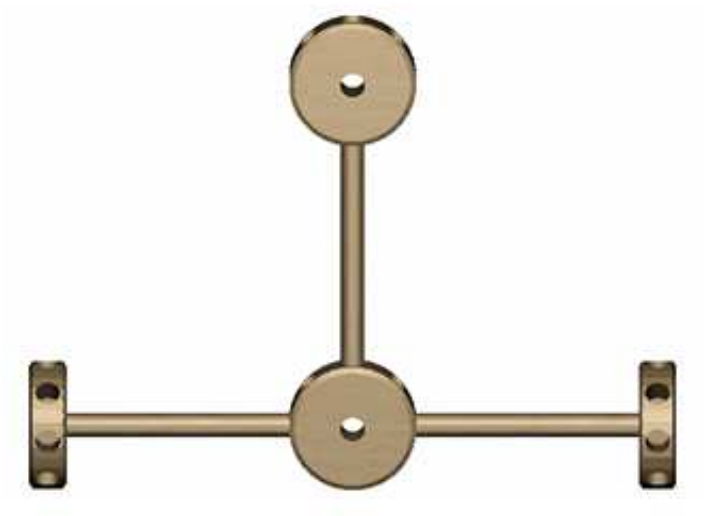

Figure A–5. Test model 2

```
spool sp1
spool sp2
spool sp3
spool sp4
stick st1
stick st2
stick st3

group g1 = connect (sp1, st1, bottom)
group g2 = connect (g1, sp2, top)
group g3 = connect (g2, st2, right)
group g4 = connect (g3, sp3, center)
group g5 = connect (g3, st3, left)
group g6 = connect (g5, sp4, center)
```

Figure A–6. Textual program for test model 2

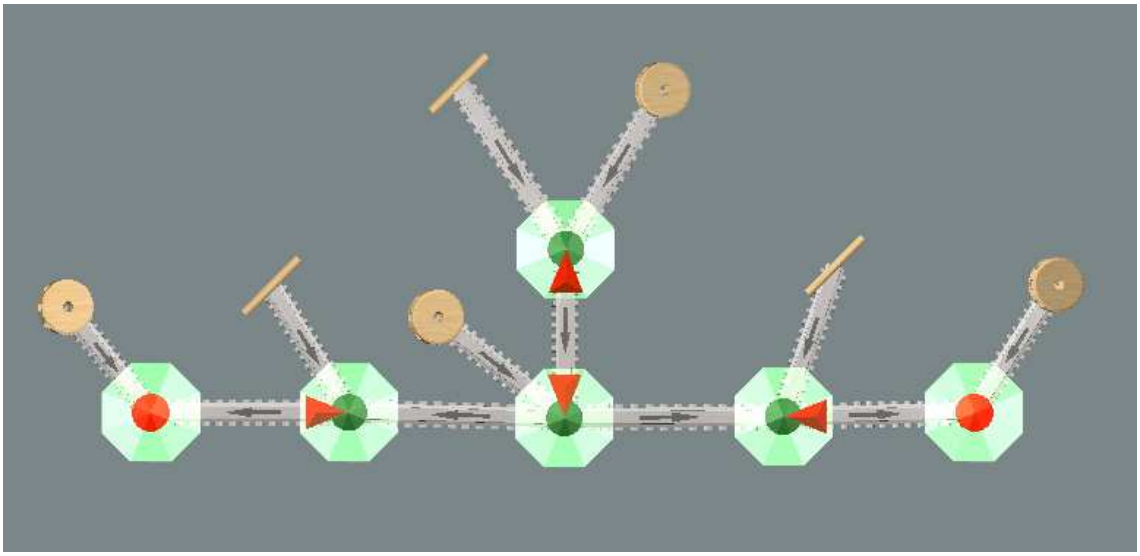

Figure A–7. Visual program for test model 2

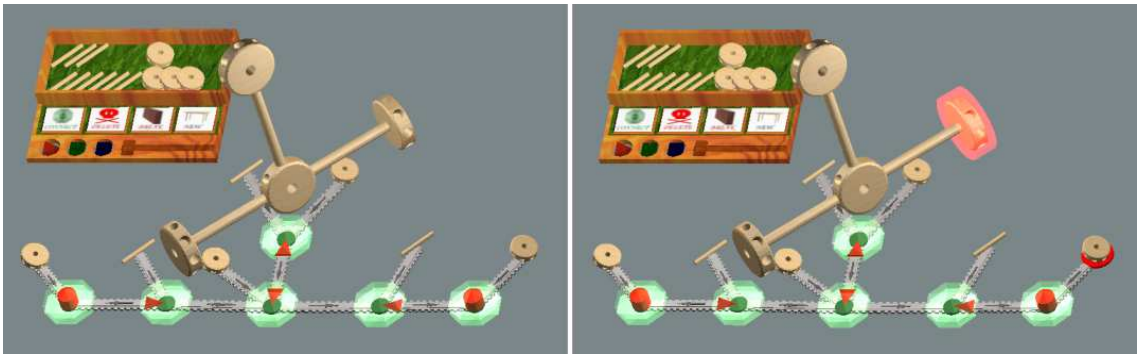

Figure A–8. Integrated model for test model 2

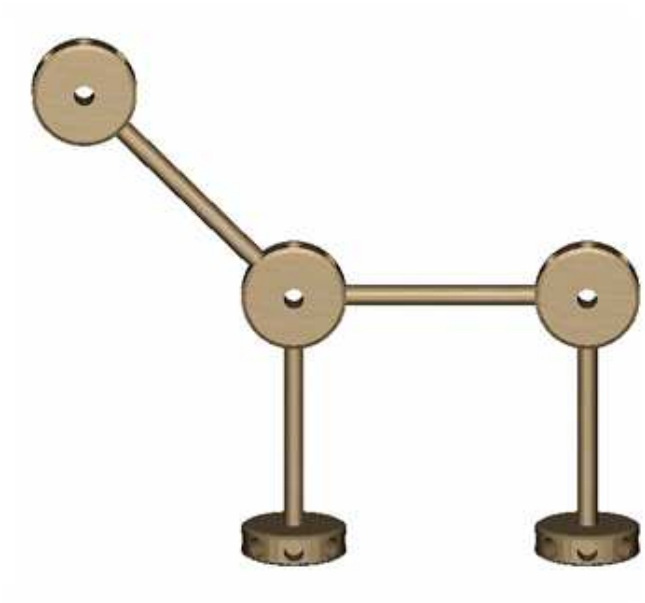

Figure A–9. Test model 3

```
spool sp1
spool sp2
spool sp3
spool sp4
spool sp5
stick st1
stick st2
stick st3
stick st4

group g1 = connect (sp1, st1, bottomRight)
group g2 = connect (g1, sp2, topLeft)
group g3 = connect (g2, st2, bottom)
group g4 = connect (g3, sp3, center)
group g5 = connect (g3, st3, right)
group g6 = connect (g5, sp4, left)
group g7 = connect (g6, st4, bottom)
group g8 = connect (g7, sp5, center)
```

Figure A–10. Textual program for test model 3

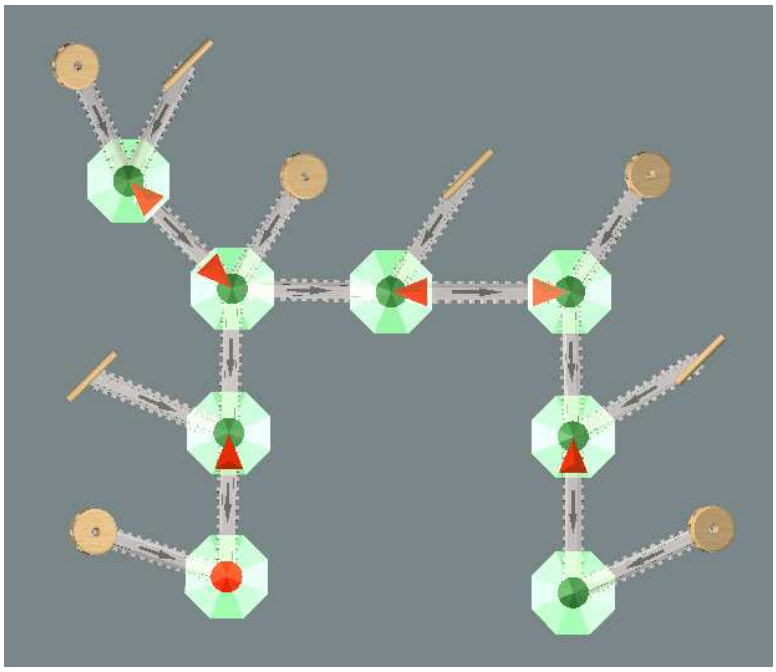

Figure A–11. Visual program for test model 3

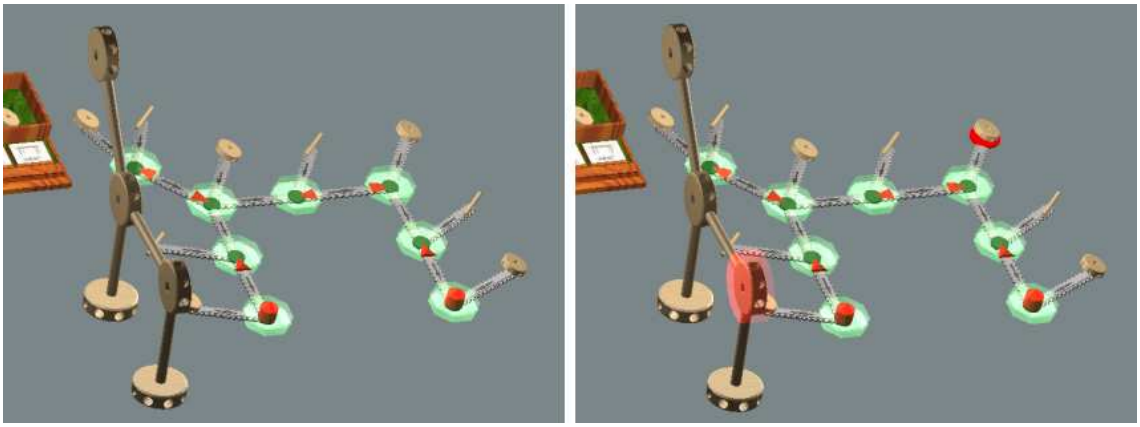

Figure A–12. Integrated model for test model 3

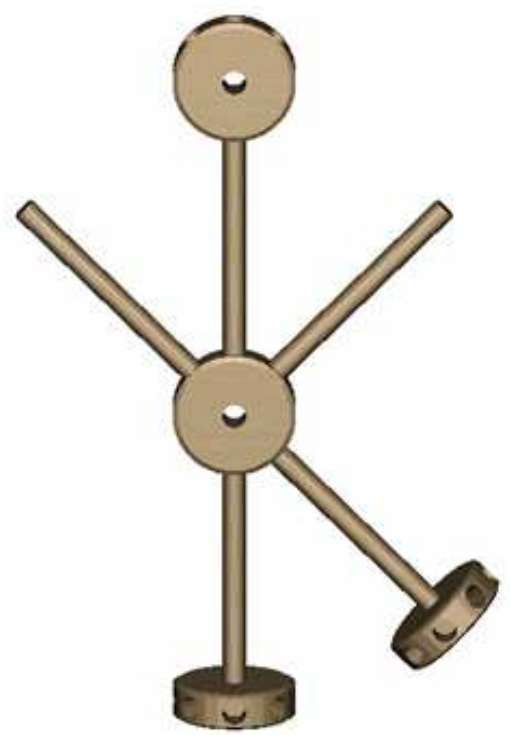

Figure A–13. Test model 4

```
spool sp1
spool sp2
spool sp3
spool sp4
stick st1
stick st2
stick st3
stick st4
stick st5

group g1 = connect (sp1, st1, bottom)
group g2 = connect (g1, sp2, top)
group g3 = connect (g2, st2, topLeft)
group g4 = connect (g2, st3, topRight)
group g5 = connect (g2, st5, bottom)
group g6 = connect (g5, sp4, center)
group g7 = connect (g2, st4, bottomRight)
group g8 = connect (g7, sp3, center)
```

Figure A–14. Textual program for test model 4

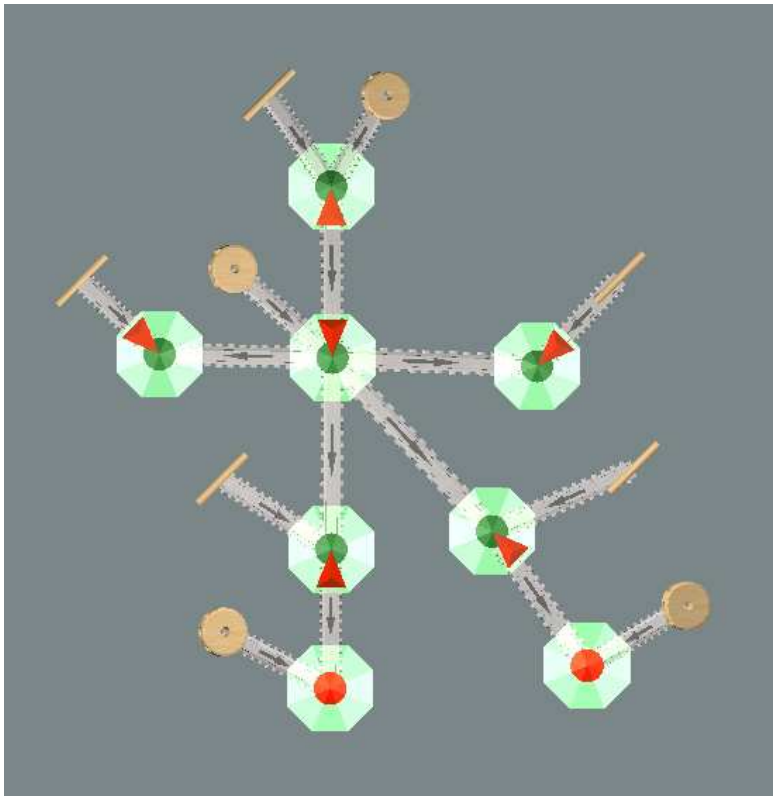

Figure A–15. Visual program for test model 4

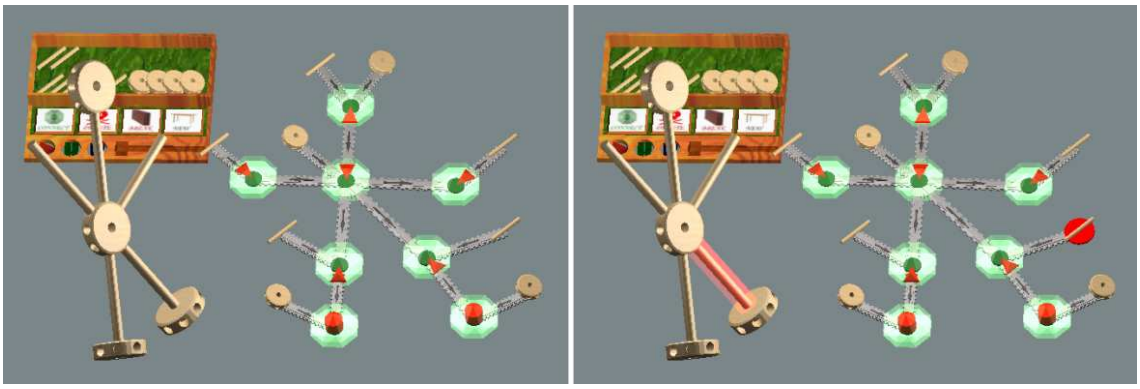

Figure A–16. Integrated model for test model 4

APPENDIX B
EMPIRICAL STUDY SURVEY DETAILS

B.1 Test Group

B.1.1 Material

1. What year are you in?
   1st 2nd 3rd 4th 5th

2. Was the programming easy?
(*No*) 1 2 3 4 5 (*Yes*)

3. Was the programming tool/language easy to use?
(*No*) 1 2 3 4 5 (*Yes*)

4. Did you enjoy programming with this tool?
(*No*) 1 2 3 4 5 (*Yes*)

5. Did unfamiliarity with the tool, if there was any, hinder your programming?
(*No*) 1 2 3 4 5 (*Yes*)

6. If you experienced textural programming, was programming using 3D icons easier than textural programming?
(*No*) 1 2 3 4 5 (*Yes*)

7. Do you think this new style of programming would motivate you for learning programming?
(*No*) 1 2 3 4 5 (*Yes*)

8. What are the points that need be considered to improve this tool further?

9. Any comment?

B.1.2 Quantitative Results

Table B–1 lists the mean of each survey question from the test group.

Table B–1. Results for the Multiple-choice Questions on the Survey

Survey Questions	Mean
What year are you in?	2.1
Was the programming easy?	4.3
Was the programming tool/language easy to use?	4.3
Did you enjoy programming with this tool?	4.1
Did unfamiliarity with the tool, if there was any, hinder your programming?	2.4
If you experienced textual programming, was programming using 3D icons easier than textual programming?	3.8
Do you think this new style of programming would motivate you for learning programming?	3.7

B.1.3 Qualitative Results

B.1.3.1 Coding

Table B–2 summarizes the responses to the open-ended questions from the test group.

Table B–2. Responses for the VPTinker Programming

	Category	Number of Responses
Comment	Easy and simple	7
	Enjoy	5
	Useful/great learning tool	4
	Interesting	3
Suggestion	Applying to other programming	5
	Displaying output progressively	4
	Confused	4
	Space/layout management	3
	Labeling objects	1

B.1.3.2 Quotes

Responses for the question *What are the points that need be considered to improve this tool further?* were categorized as follows:

Confusion.

- It was a little difficult to figure out how to connect two sticks to one spool.
- Better description of connection rules in video.
- I think I might have connected the things wrong. It would be easier if connections were put between every part.
- More specific about connection rules.

Show output.

- Show the 3D object or model.
- A preview window which you can see your progress along the way. Or a text editing window which you can manually write code, but still keep it visual.
- If possible, have an image of the Tinker generated after every actions at the top corner.
- See the process as you work on it. Meaning, see the results as you built it.

Space.

- The layouts of programs can get very cluttered and confusing if they are large. Perhaps a way to organize it easily would help.
- Space constraints–as the program becomes more complex, the visual programming tool may have difficulty showing a large and clear view of the program.
- Movement of groups in the assembly area as opposed to moving individual pieces to create more space.

Easy and simple.

- Nothing. Straight forward easy to use.
- Better graphics possibly? Otherwise very helpful and easy to understand.
- I have no real program with textual programming as it stands. Nothing could really made it more simple.

Application.

- More knowledge of the tool. What else could this be used for in a real–life situation.
- Other uses than Tinkertoys.
- I want more objects other than spool and stick.

Labeling.

- Labeling so you wouldn't get lost along the way of your program.

Instruction.

- Interactive help center
- Better instruction
- Examples of actual data that would fit the model
- More descriptions and examples

Responses for the question *Any comment?* were categorized as follows:

Interesting.

- Interesting way of programming
- This is a very interesting set up. While graphically oriented, it is based more on process than product, as a result, the program tends to look nothing like the final construct, as one would expect from most graphical programs, but instead, resembles on assembly line floor plan.
- Very interesting to use.

Easy.

- Very easy, intuitive model for programming. No skill required. Perhaps a goal should be to make the image look like the model that is being constructed?
- This type of programming is a lot easier than textual programming.
- Very intuitive however I got rest 3 incorrect. Easy to use but unsure if I used it correctly.
- I also ran this experiment using textual environment. I think for thinker toy visual was easier to use and understand because test 1–4 were presented

visually by drawings. So by using a visual environment it was easier to manipulate the programming match the picture on the previous page. If the test were stated in words, I believe it would be easier to use the textual environment. But since tinker toys are easily understood with pictures, a visual programming environment is ideal.

Enjoy.

- I enjoyed it
- Cool use of blender
- Nice idea
- Great program
- Well made and thought out program

Learning.

- I am already motivated to learn programming I like the tool.
- Great learning tool.
- This programming tool may be helpful to people who have trouble with abstract thinking, but it would take time to learn.
- Overall, I think this is a useful program. Other programming languages like Java, C, and etc. make us to remember number of codes. It is easier to build a program by using 3D icons.

Application.

- That would be good if this language apply to solve problem.
- I doubts its application to big projects.

Negative.

- There is no challenge in icon programming. With text, it's a puzzle to solve.
- I really can't say. I am biased towards text. Nothing visual would appeal to me.

B.2 Controls group

B.2.1 Material

1. What year are you in?
   1st 2nd 3rd 4th 5th

2. Was the programming easy?
(*No*) 1 2 3 4 5 (*Yes*)

3. Was the programming tool/language easy to use?
(*No*) 1 2 3 4 5 (*Yes*)

4. Did you enjoy programming with this tool?
(*No*) 1 2 3 4 5 (*Yes*)

5. Did unfamiliarity with the tool, if there was any, hinder your programming?
(*No*) 1 2 3 4 5 (*Yes*)

6. Any comment?

B.2.2 Quantitative Results

Table B–3 lists the mean of each survey question from the control group.

Table B–3. Results for the Multiple-choice Questions on the Survey

Survey questions	Mean
What year are you in?	2.2
Was the programming easy?	4.3
Was the programming tool/language easy to use?	4.1
Did you enjoy programming with this tool?	4.1
Did unfamiliarity with the tool, if there was any, hinder your programming?	1.3

B.2.3 Qualitative Results

B.2.3.1 Coding

Table B–4 summarizes the responses to the open-ended question from the control group.

Table B–4. Responses for the Textual Programming

Category	Number of Responses
Confused	8
Enjoy	8
Easy and simple	6

B.2.3.2 Quotes

Responses for the question *Any comment?* were categorized as follows:

Confusion.

- Somewhat hard to decipher the coding when looking for a bug/mess-up. Does program depend on the order you write to produce final product? like when it shows final the last thing put on will be the one touching the "ground'?

- The programming language was too easy or rather too vague, needs to be more specific.

- There can be some confusion as to what the groups we are creating refer to, especially when 2 spools and 1 stick are used. Than the continuous building upon those might come out completely different than anticipated.

- Unclear as to how connect works.

- It was hard to decipher whether what was being connected to what.

- The programming was not very difficult to use, but you need to be extremely careful with how you position the spools and sticks. I think I messed up mostly because I grouped things incorrectly.

- I thought the connect method was assigning new connections to the group. But I did not realize I was re–initializing the group connection. Perhaps, the problem would be solve if instead of saying group g1 = connection() I would say g1.connect, but I don't know the syntax of the grogram that well. I know creating more groups would have solve the problem.

- It was very unclear the order to build the groups so as to connect everything properly. When it was necessary to branch off more than two times from the same spool it was confusing on what to do. A suggestion of mine would be to ignore the group portion and simply tell it to connect, for instance problem 2 would be: After declaring the 4 spools and 3 sticks)
  connect(sp1,st1,bottom)
  connect(sp3,st1,top)
  connect(sp3,st2,left)
  connect(sp3,st3,right)
  connect(st2,sp2,center)
  connect(st3,sp4,center)
  I feel this way of programming is easier to perform and understand.

Enjoy.

- Nice
- Cool program
- That was new
- Practice makes perfect
- I like this better than Java.
- I was excited, and made stupid mistakes. Really cool.
- More enjoyable than programming with standard codes like Java or c++.

Simple.

- Nice. simple language. Works well
- Very simple language to learn and implement
- Very simple language to understand
- Very simple and easy to use program

Easy.

- Very easy to use and cool to see the graphics do what you want.

- Very good program, easy to use. I messed up an test 4. I guess I got confused because there were so many actions taking place on that middle spool. I am sure it would be easy to correct once the computer said there was an error.

  Interesting.

- Very interesting project

  Test materials.

- Larger models became exponentially more difficult.

REFERENCES

[1] I. Sutherland, "Sketchpad: A Man Machine Graphical Communication System," PhD dissertation, Department of Electrical Engineering, MIT, 1963.

[2] D. C. Smith,jd "PYGMALION: A Creative Programming Environment," PhD dissertation, Department of Computer Science, Stanford University, 1975.

[3] E P. Glinert (ed.), *Visual Programming Environments Paradigms and Systems*, 1990, IEEE Computer Society Press: Los Alamitos, CA.

[4] N. C. Shu, *Visual Programming*, 1988, Van Nostrand Reinhold Company: New York, NY.

[5] S. Chang, *Principles of Visual Programming Systems*, 1990, Prentice–Hall: Englewood Cliffs, NJ.

[6] P. E. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics*, 1988, 22(4): 103–111.

[7] K. Kahn and V. Saraswat, "Complete Visualizations of Concurrent Programs and Their Executions," in Proceedings of the IEEE Visual Language Workshop, 1990, Skokie, IL, 7–15.

[8] M. Duecker, G. Lehrenfeld, W. Mueller, and C. Tahedl, "A Generic System for Interactive Real–Time Animation," in Proceedings of IEEE International Conference on Engineering of Computer–Based Systems, 1997, Monterey, CA, 263–270.

[9] R. Stiles and M. Pontecorvo, "Lingua Graphica: A Visual Language for Virtual Environments," in IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1992, Seattle, WA, 255 – 227.

[10] C. Geiger, W. Mueller, and W. Rosenbach, "SAM – An Animated 3D Programming Language," in 1998 IEEE Symposium on Visual Languages, 1998, Halifax, Canada, 228–235.

[11] K. Yamamoto, "3D–Visualan: A 3D Programming Language for 3D Applications," Pacific Workshop on Distributed Multimedia Systems, 1996, Hong Kong, 199–206.

[12] H. Koike, T. Takada, and T. Masui, "VisuaLinda: A Framework for Visualizing Parallel Linda Programs," in Proceeding of 1997 IEEE Symposium on Visual Languages, 1997, Capri, Italy, 174–180.

[13] B. A. Calloni and D. J. Bagert, "Iconic Programming in BACCII vs. Textual Programming: Which is a Better Learning Environment?," in Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education, 1994, Phoenix, Arizona, 188–192.

[14] T. R. G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs," Sixth European Conference on Cognitive Ergonomics, 1992, Balatonfured, Hungary, 167–180.

[15] M. G. Williams, H. Villegas, and J. N. Buehler, "Appropriateness of Graphical Program Representations for Training Applications," in Proceedings of the CHI '96 Conference on Human Factors in Computing Systems, 1996, Vancouver, BC, Canada, 91–92.

[16] R. Pandey and M. M. Burnett, "Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study," in Proceedings of the IEEE Symposium on Visual Languages, 1993, Bergen, Norway, 344–351.

[17] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation," in Proceedings of the 22nd International Conference on Software Engineering, 2000, Kyoto, Japan, 230–239.

[18] National Instruments Corporation, LabVIEW: 20 Years of Innovation, 2006, Available from URL: http://www.ni.com/labview, Site last visited July 2006.

[19] AccuSoft, VisiQuest, 2006, Available from URL: http://www.accusoft.com/products/visiquest, Site last visited July 2006.

[20] The MathWorks, Inc., Simulink, 2006, Available from URL: http://www.mathworks.com/products/simulink, Site last visited July 2006.

[21] SciTegic, 2006, SciTegic, Available from URL: http://www.scitegic.com, Site last visited July 2006.

[22] J. F. Hopkins and P. A. Fishwick, "The Rube Framework for Software Modeling and Customized 3–D Visualization," *Journal of Visual Languages and Computing*, 2003, 14(1): 97–117.

[23] F. Fogarty, J. Forlizzi, and S. E. Hudson, "Aesthetic Information Collages: Generating Decorative Displays That Contain Information," in Proceedings of the ACM Symposium on User Interface Software Technology, 2001, Orlando, Florida, 141–150.

[24] N. Tractinsky, "Toward the Study of Aesthetics in Information Technology," in the Twenty–Fifth International Conference on Information Systems, 2004, Washington, D.C., 771–780.

[25] R. Lavie and N. Tractinsky, "Assessing Dimensions of Perceived Visual Aesthetics of Web Sites," *International Journal of Human–Computer Studies*, 2004, 60(3): 269–298.

[26] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, 4th edition*, 2003, Addison–Wesley Publishing Company: Boston, MA.

[27] D. Hearn and M. P. Baker, *Computer Graphics With Opengl 3rd Edition*, 2003, Prentice Hall:Upper Saddle River, NJ.

[28] Sun Microsystems, Inc., Java 3D API, 2006, Available from URL: http://java.sun.com/products/java–media/3D, Site last visited July 2006.

[29] M. Q. W. Baldonado, A. Woodruff, and A. Kuchinsky, "Guidelines for Using Multiple Views in Information Visualization," in Proceedings of the Working Conference on Advanced Visual Interfaces, 2000, Palermo, Italy, 110–119.

[30] S. F. Roth, P. Lucas, J. A. Senn, C. C. Comberg, M. B. Burks, P. J. Stroffolino, J. A. Kolojejchick, and C. Dunmire, "Visage: A User Interface Environment for Exploring Information," in Proceedings of the IEEE Information Visualization Symposium, 1996, San Francisco, CA, 3–12.

[31] C. North and B. Shneiderman, "Snap–Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata," in Proceedings of ACM Advanced Visual Interfaces, 2000, Palermo, Italy, 128–135.

[32] C. R. Santos and P. Gros, "Multiple Views in 3D Metaphoric Information Visualization," in Proceedings of Sixth International Conference on information Visualisation, 2002, London, England, UK, 468–473.

[33] A. Buja, J. A. McDonald, J. Michalak, and W. Stuetzle, "Interactive Data Visualization Using Focusing and Linking," in Proceedubgs of IEEE Visualization, 1991, San Diego, CA, 156–163.

[34] R. Kosara, G. N. Sahling, and H. Hauser, "Linking Scientific and Information Visualization with Interactive 3D Scatterplots," in Proceedings of the 12th International Conference in Central Europe on Computer Graphics, 2004, Plzen-Bory, Czech Republic, 133–140.

[35] R. Kosara, H. Hauser, and D. L. Gresh, "An Interaction View on Information Visualization," in EUROGRAPHICS, 2003, Granada, Spain, 123–137.

[36] P. A. Fishwick, "Toward an Integrative Multimodeling Interface: A Human–Computer Interface Approach to Interrelating Model Structures," *Simulation*, 2004, 80(9): 421–432.

[37] M. Park and P. A. Fishwick, "A Methodology for Integrative Multimodeling: Connecting Dynamic and Geometry Models," in Proceedings of Enabling Technology for Simulation Science, Part of SPIE Aerosense '04 Conference, 2004, Orlando, FL, 9–17.

[38] J. D. Johnson and G. W. Kenney, "Implementation Issues for a Source Level Symbolic Debugger (Extended Abstract)," in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High–Level Debugging, 1983, Pacific Grove, CA, 149–151.

[39] M. S. Johnson, "Some Requirements for Architectural Support of Software Debugging," in Symposium on Architectural Support for Programming Languages and Operating Systems, 1982, Palo Alto, CA, 140–148.

[40] H. Lieberman, "A Three–Dimensional Representation for Program Execution," in Proceedings of the 1989 Workshop on Visual Languages, 1989, Los Alamitos, CA, 111–116.

[41] T. Okamura, B. Shizuki, and J. Tanaka, "Execution Visualization and Debugging in Three–Dimensional Visual Programming," in Proceedings of the 8th International Conference on Information Visualisation, 2004, London, England, 167–172.

[42] L. McIver and D. M. Conway, "Seven Deadly Sins of Introductory Programming Language Design," in Proceedings of Software Engineering: Education and Practice, 1996, Los Alamitos, CA, 309–316.

[43] N. Cunniff, R. P. Taylor, and J. B. Black, "Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal," *ACM SIGCHI Bulletin*, 1986, 17(4): 175–182.

[44] T. R. G. Green and R. Navarro, "Programming Plans, Imagery, and Visual Programming," in Proceedings of INTERACT '95, 1995, Lillehammer, Norway, 139–144.

[45] R. Navarro and J. J. Cañas, "Mental Representation and Imagery in Program Comprehension," in Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group, 1999, Leeds, UK.

[46] G. A. Lee, G. J. Kim, and C. Park, "Modeling Virtual Object Behavior within Virtual Environment," in Proceedings of the ACM symposium on Virtual reality software and technology, 2002, Hong Kong, China, 41–48.

[47] F. V. Reeth, K. Coninx, S. D. Backer, and E. Flerackers, "Realizing 3D Visual Programming Environments Within a Virtual Environment," *Computer Graphics Forum*, 1995, 14(3): 361–370.

[48] T. Oshiba and J. Tanaka, "3D–PP: Three–Dimensional Visual Programming System," in 1999 IEEE Symposium on Visual Languages, 1999, Tokyo, Japan, 13–16.

[49] M. F. Sanner, D. Stoffler, and A. J. Olson, "Viper, a Visual Programming Environment for Python," in the Tenth International Python Conference, 2002, Alexandria, VA, 103–115.

[50] C. R. Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. P. Paris, "Metaphor–Aware 3D Navigation Information Visualization," in IEEE Symposium on Information Visualization 2000, 2000, Salt Lake City, UT, 155–165.

[51] C. R. Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. P. Paris, 'Mapping Information onto 3D Virtual Worlds," in Proceedings of the International Conference on Information Visualization, 2000, London, UK, 379–386.

[52] J. Rekimoto and M. Green, "The Information Cube: Using Transparency in 3D Information Visualization," in Proceedings of the Third Annual Workshop on Information Technologies and Systems, 1993, Orlando, FL, 125–132.

[53] J. Rilling and S. P. Mudur, "On the Use of Metaballs to Visually Map Source Code Structures and Analysis Results onto 3D Space," in Proceedings of Ninth Working Conference Reverse Engineering, 2002, Richmond, Virginia, 299–308.

[54] T. Panas, R. Berrigan, and J. C. Grundy, "A 3D Metaphor for Software Production Visualization," in the Seventh International Conference on Information Visualization, 2003, London, UK, 314–319.

[55] D. Ploix, "The Zeugma Programming Environment," Rapport de Recherche, Department d'Informatique, RR980123, Universite Paris, J1998.

[56] D. Ploix, "Building Program Metaphors," in Proceedings of Psychology of Programming Interest Group Workshop, 1996, Matlock, UK, 125-129.

[57] D. Ploix, "Analogical Representation of Programs," in Proceedings of First International Workshop on Visualizing Software for Understanding and Analysis, 2002, Paris, France, 61-69.

[58] C. Knight and M. Munro, "Virtual but Visible Software," in Proceedings of International Conference on Information Visualisation, 2000, London, England, 198–205.

[59] J. F. Hopkins and P. A. Fishwick, "Exploiting an Agent–Based Metaphor in Software Visualization Using the Rube System," *Journal of Visual Languages and Computing*, 2003, 14(1): 97–117.

[60] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software Landscapes: Visualizing the Structure of Large Software Systems," in Proceedings of the Symposium on Visualization, 2004, Konstanz, Germany, 261–266.

[61] Blender, Blender: Open Source 3D Graphics Creation, 2006, Available from URL: http://blender.org/cms/Home.2.0.html, Site last visited July 2006.

[62] G. Lakoff and M. Johnson, *Metaphors We Live By*, 1980, The University of Chicago Press: Chicago, IL.

[63] A. Pirhonen and S. Brewster, "Metaphors and Imitation," Workshop on Integrating Multimedia, Metaphors, and Multimodality, 8th PC–HCI, 2001, Patras, Greece, 27–32.

[64] E. Harslem and L. E. Nelson, "A Retrospective on the Development of Star," in Proceedings of the Sixth International Conference on Software Engineering, 1982, Tokyo, Japan, 377–383.

[65] R. M. Meyer and T. Masterson, "Towards a Better Visual Programming Language: Critiquing Prograph's Control Structures," *The Journal of Computing in Small Colleges*, 2000, 15(5): 181–193.

[66] A. F. Blackwell and T. R. G. Green, "Does Metaphor Increase Visual Language Usability?," in Proceedings of the 1999 IEEE Symposium on Visual Languages VL'99, 1999, Tokyo, Japan, 246–253.

[67] P. A. Fishwick, "Enhancing Experiential and Subjective Qualities of Discrete structure Representations with Aesthetic Computing," *Journal of Visual Languages and Computing*, 2005, 16(5): 406–427.

[68] W. H. Mayall, *Principles in Design*, 1979, The Design Council:London, UK.

[69] D. A. Norman, *Emotional Design: Why We Love (or Hate) Everyday Things*, 2004, Basic Books: New York, NY.

[70] P. A. Fishwick (ed.), *Aesthetic Computing*, MIT Press, 2005.

[71] M. L. Scott, *Programming Language Pragmatics*, 2000, Morgan Kaufmann: San Diego, CA.

[72] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, 1995, Prentice–Hall: Upper Saddle River, NJ.

[73] G. Gordon, *System simulation*, 1969, Prentice-Hall: Englewood Cliffs, NJ.

[74] P. J. Kiviat et al, "SIMSCRIPT: A Simulation Programming Language,"1973, CACI.

[75] R. E. Nance, "A History of Discrete Event Simulation Programming Languages," *ACM SIGPLAN Notices*, 1993, 28(3): 149–175.

[76] J. Banks and J. S. Carson, "Introduction to Discrete–Event Simulation," in Proceedings of the 1986 Winter Simulation Conference, 1986, Washington, D.C., 17–23.

[77] A. Bloss, "A Functional Approach to Simulation Programming," in Proceedings of the 1990 Winter Simulation Conference, 1990, New Orleans, LJ, 214–219.

[78] M. Park and P. A. Fishwick, "Integrating Dynamic and Geometry Model Components through Ontology-Based Inference," *Simulation: Transactions of The Society for Modeling and Simulation International*, 2005, 81(12): 795-813.

[79] H. Shim, P. A. Fishwick, "A Customizable Approach to Visual Programming Using Dynamic Multimodeling," in Proceedings of Enabling Technology for Simulation Science, Part of SPIE Aerosense '04 Conference, 2004, Orlando, FL, 1931–1938.

[80] V. Fix, S. Wiedenbeck, and J. Scholtz, "Mental Representation of Programs by Novices and Experts," in Proceedings of the SIGCHI conference on Human factors in computing systems, 1993, Amsterdam, The Netherlands, 74–79.

[81] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, 1989, 21(4): 593–622.

[82] P. A. Fishwick, J. Lee, M. Park, and H. Shim, "RUBE: A Customized 2D and 3D Modeling Framework for Simulation," in Proceedings of the 2003 Winter Simulation Conference, 2003, New Orleans, LA, 755–762.

[83] J. Lee and P. A. Fishwick, "A Dynamic Exchange Language Layer for Rube," in Proceedings of Enabling Technology for Simulation Science, Part of SPIE Aerosense '02 Conference, 2002, Orlando, FL, 359–366.

[84] M. Park and P. A. Fishwick, "SimPackJ/S: A Web–Oriented Toolkit for Discrete Event Simulation," in Proceedings of Enabling Technology for Simulation Science, Part of SPIE Aerosense '02 Conference, 2002, Orlando, FL, 348–358.

[85] J. A. Miller, P. A. Fishwick, S. J. E. Taylor, P. C. Benjamin, and B. K. Szymanski, "Research and Commercial Opportunities in Web–Based Simulation," *Simulation Practice and Theory*, 2001, 9(1): 55–72.

[86] E. H. Page, "Web–Based Simulation: Revolution or Evolution?," *ACM Transactions on Modeling and Computer Simulation*, 2000, 10(1): 3–17.

[87] K. L. Damkjer, "Architecting RUBE Worlds: A Methodology for Creating Virtual Analog Devices as Metaphorical Representations of Formal Systems," Master thesis, Department of Computer & Information Science & Engineering, University of Florida, 2003.

[88] P. A. Fishwick and J. A. Miller, "Ontologies for Modeling and Simulation: Issues and Approaches," in Proceedings of the 2004 Winter Simulation Conference, 2004, Washington, DC, 259–264.

[89] P. A. Fishwick, "Toward an Integrative Multimodeling Interface: A Human–Computer Interface Approach to Interrelating Model Structures," *The "Grand Challenges" special issue of SCS Transactions on Modeling and Simulation*, 2004, 80(9): 421–432.

[90] Ventana Systems, Inc., Vensim, 2005, Available from URL: http://www.vensim.com, Site last visited July 2006.

[91] Isee systems, inc., Stella, 2006, Available from URL: http://www.iseesystems.com, Site last visited July 2006.

[92] Blender, GameBlender, 2006, Available from URL: http://www.blender3d.org/cms/GameBlender.576.0.html, Site last visited July 2006.

[93] C. Ware, G. Franck, M. Parkhi, and T. Dudley, "Layout for Visualizing Large Software Structures in 3D," in Proceedings of VISUAL'97, 1997, San Diego, CA, 215–223.

[94] E. Kleiberg, H. V. E. Wetering, and J. J. Wijk, "Botanical Visualization of Huge Hierarchies," in Proceedings of IEEE Symposium on Information Visualization (InfoVis'2001), 2001, San Diego, CA, 87–94.

[95] T. Dwyer, "Three Dimensional UML Using Force Directed Layout," in Proceedings of the Australian Symposium on Information Visualisation, 2001, Sydney, Australia, 77–85.

[96] C. Breiteneder, H. Eidenberger, G. Fiedler, and M. Raab, "Lookmark: A 2.5D Web Information Visualization System," in Information and Communication Technology: First Eurasian Conference, 2002, Shiraz, Iran, 93–101.

[97] G. Robertson, M. Czerwinski, K. Larson, D. C. Robbins, D. Thiel, and M. Dantzich, "Data Mountain: Using Spatial Memory for Document Management," in Symposium on User Interface Software and Technology, 1998, San Francisco, CA, 153–162.

[98] M. Tavanti and M. Lind, "2D vs 3D, Implications on Spatial Memory," in Proceedings of IEEE InfoVis 2001 Symposium on Information Visualization, 2001, Washington, DC, 139–145.

[99] M. H. Brown, *Algorithm Animation*, 1988, MIT Press: Cambridge, MA.

[100] M. H. Brown, "Exploring Algorithms Using Balsa–II," *IEEE Computer*, 1988, 21(5): 14–36.

[101] J. T. Stasko, "XTANGO : A Framework and System for Algorithm Animation," *IEEE Computer*, 1990, 23(9): 27–39.

[102] M. H. Brown, "Zeus: A System for Algorithm Animation and Multi-View Editing," *Computer Graphics*, 1992, 18(3): 177–186.

[103] M. D. Byrne, R. Catrambone, and J. T. Stasko, "Do Algorithm Animations Aid Learning?," Technical Report GIT–GVU–96–18, GVU Center, Georgia Institute of Technology, Atlanta, GA, 1996.

[104] J. T. Stasko, A. Badre, and C. Lewis, "Do Algorithm Animations Assist Learning? An Empirical Study and Analysis," in Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, 1993, Amsterdam, Netherlands, 61–66.

[105] A. Lawrence, A. Badre, and J. T. Stasko, "Empirically Evaluating the Use of Animations to Teach Algorithms," in Proceedings of the 1994 IEEE Symposium on Visual Languages, 1994, Saint Louis, MO, 48–54.

[106] R. Rasala, V. Proulx, and H. Fell, "From Animation to Analysis in Introductory Computer Science," in Proceedings of ACM Computer Science Conference, 1994, Phoenix, AZ, 61–65.

[107] J. T. Stasko, "Using Student–Built Algorithm Animations as Learning Aids," in Proceedings of the twenty–eighth SIGCSE technical symposium on Computer science education, 1997, San Jose, CA, 25–29.

[108] T. L. Naps, R. Fleischer, M. McNally, G. Röbling, C. Hundhausen, S. Rodger, V. Almstrum, A. Korhonen, J. Velzquez–Iturbide, W. Dann, and L. Malmi, "Exploring the Role of Visualization and Engagement in Computer Science Education," *ACM SIGCSE Bulletin*, 2003, 35(2): 131–152.

[109] W. Citrin, R. Hall, C. Santiago, and B. Zorn, "Addressing the scalability problem in visual programming throughcontainment, zooming and fisheyeing," in Proceedings of the IEEE Aerospace Conference, 1998, Snowmass, CO, 189–202.

[110] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, and S. Yang, "Scaling up Visual Programming Languages," *IEEE Computer*, 1995, 28(3): 45–54.

[111] J. Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics*, 1986, 5(2): 110–141.

[112] D. Gentner, "The Mechanisms of Analogical Learning," in *S. Vosniadou and A. Ortony (eds.), Similarity and Analogical Reasoning*, 1989, Cambridge University Press: Cambridge, MA, 199–241.

[113] M. C. Heopfl, "Choosing Qualitative Research: A Primer for Technology Education Researchers," *Journal of Technology Education*, 1997, 9(1): 47-63.

[114] H. R. Bernard, "Qualitative Data, Quantitative Analysis," *Cultural Anthropology Methods Journal*, 1996, 8(1) 9-11.

[115] T. R. G. Green, "Cognitive Dimensions of Notations," in *A. Sutcliffe and L. Macaulay (eds.), People and Computers V*, 1989, Cambridge University Press: Cambridge, MA, 443-460.

[116] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing*, 1996, 7(2) 131–174.

[117] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roes, and A. Wong, "Cognitive Dimensions of Notations: Design tools for cognitive technology," in the Fourth International Conference on Cognitive Technology, 2001, Warwick, UK, 325–341.

[118] J. P. Gee, "Learning by Design: Good Video Games as Learning Machines," *E-Learning*, 2005, 2(1): 5–16.

[119] Side Effects Software Inc., Houdini, 2006, Available from URL: http://www.sidefx.com, Site last visited July 2006.

BIOGRAPHICAL SKETCH

Hyunju Shim is a Ph.D. student in computer and information science and engineering at the University of Florida. She received her M.S. in computer and information science and engineering from the University of Florida in 2003. Her research interests include computer modeling and simulation, visual programming, and software visualization. Her email address and Web addresses are <hshim@cise.ufl.edu> and <http://www.cise.ufl.edu/∼hshim>.