Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

PROGRAMMING USING DYNAMIC SYSTEM MODELING VIA A 3D-BASED MULTIMODELING FRAMEWORK

Hyunju Shim
Paul Fishwick

Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611, U.S.A.

ABSTRACT

We propose a new approach to visual programming which adopts principles and elements from dynamic multimodeling for structured procedural programming, especially graphics programming. Unlike most traditional visual programming languages which simply replace syntactic parts of program with graphical objects, we applied the principles of dynamic model types in modeling and simulation to create program models and execute/simulate them. With this approach, computer programs are constructed by visual modeling instead of textual writing. The motivation for a method using dynamic model types in graphics programming is also tied to several emerging research areas: novice user interfaces, programming visualization, customized icons, and a broader view of aesthetics within programming. Metaphoric icons are extensively used for the visual representation of program model elements. *Rube*, a Web- and XML-based modeling and simulation framework, provides the necessary environment for the construction, visualization and execution of program models.

1 INTRODUCTION

The first research on programming by demonstration was Pygmalion (Smith 1975). The basic idea was that programming systems should support visual and analogical aspects of creative thought and that programming should be less tedious. Ever since the first visual programming system was introduced, visual programming has been spotlighted and a good volume of principles and methodologies have been developed. While most of visual programming introduced until '80s were 2D, 3D visual programming is getting increasing interest from a wide range of domains provided by the availability of low cost hardware and software for 3D graphics.

There has been much research in visual programming regarding the generation of effective representations of programs for different programming languages or paradigms. However, issues such as 3D customizable icons, stimulation of user's creativity and aesthetic perspectives of visualization are often neglected among visual programming communities. Our research proposes a new approach to visual programming that unifies modeling with programming while leveraging 3D customizable icons that stimulates user's creativity and aesthetic perspectives. This approach to programming has particular utility in education, where creativity has been shown to increase subject interest. Unlike most visual programming languages that visually mimic the behavior of textual programs, we applied dynamic model types such as Functional Block Model (FBM) and Finite State Machine (FSM) in the construction and execution of program models 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.

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. *RUBE* makes use of a 3D open source tool Blender to provide a modeling environment for users. Users can use pre-existing program modules and icons or create their own modules and icons in *RUBE*.

The primary innovation of our work is the following:

- A customized approach to modeling, enabling novices to learn modeling through a high degree of creativity. We express this creativity in our model representations through the use of personalized 3D, rather than 2D, icons
- An approach to programming that leverages a set of "off the shelf" dynamic model types tradition-

ally associated with the simulation community, without necessarily attempting to institute a standard: data flow graphs, event graphs, and Petri nets, for example, have their own approaches to visual syntax and application
- A holistic environment where the program syntax and semantic behavior are situated within the same 3D virtual space.

2 RELATED WORK

ConMan (Haeberli 1988) is a 2D high-level visual language that lets users dynamically build and modify graphics applications. A data flow metaphor is used in ConMan and users construct and modify complete applications by creating components that are interconnected via input and output ports. Developers are encouraged to break monolithic applications into functional components that communicate with each other using high level data structures.

Lingua Graphica (Stiles and Pontecorvo 1992) is 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, dataflow 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.

SAM (Geiger, Mueller, and Rosenbach 1998) is a visual 3D programming language, visualization, and environment for parallel systems specification and animation. A SAM program is mainly given as a set of 3D objects and there is no separate textual description. 3D 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. SAM 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 to give a more realistic environment.

3D-PP (Oshiba and Tanaka 1999) applies the direct manipulation of operations to the 3D program elements. 3D-PP is based on the concurrent logic programming language 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, by using the nesting level filtering and the double-click browsing improves the spatial problem in visual programming caused by a small screen.

3D-Visualan (Yamamoto 1996) 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. Using 3D-Visulan, programmers can express 3D worlds by 3D worlds.

Pictorial Janus (PJ) (Kahn and Saraswat 1990) is a 2D visual programming language for the parallel logical programming. Programs in PJ are drawings where the execution is defined as the animation of the drawings. A PJ program is given by the composition of closed contours and directed and undirected connections between them. Objects of PJ are constants, list elements, links, functions, agents, and rules. Agents in 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. JIM (Janus In Motion) provides interactive PJ specification and animation environment.

PiP (Lee, Kim, and Park 2002) is a virtual environment system in which a user can create, modify, test, and save object behaviors. PiP extends the visual programming approach to 3D multimodal programming, representing the program or object behavior in forms using 3D visual objects and programming them through manipulation in 3D space. The most typical behaviors of objects are specified by demonstration. Using PiP, users can program or specify the motion or animation intuitively through better understanding of direct programming in 3D space.

3 VISUAL PROGRAMMING VIA MODELING

3.1 Programming versus Modeling

While programming is the process of creating a computer program to solve problems with instructions that the computer can interpret, modeling in general refers to the creation of the representation for a certain system. The common interest of programming and simulation modeling is to solve problems in real life using a computer. However, 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 and to help human decision (Fishwick 1995). The field of computer modeling and simulation has developed different models of computational paradigms, and execution modes from those in programming (Banks and Carson 1986, Nance 1993).

There also has been much research relating programming with modeling. In 1965, Sutherland created a dataflow language that allows for visual creation, debugging, and execution of dataflow diagrams (Sutherland 1963). SIMULA is the first object-oriented language and a good example of how modeling and simulation principles can improve programming. Bloss (1990) 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. While not every principle and element of programming can be modeled, there are cases where dynamic model types in simulation can improve programming in terms of its expressiveness and understandability (Shim and Fishwick 2004). The rest of this paper elucidates how dynamic model types can be used and combined together to model program constructors.

3.2 Mapping Program to Model Elements

In this section, issues that are related to the construction of visual programming models for structured procedural programs using dynamic model types, will be discussed in detail.

3.2.1 Representation of 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, 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 (Yamamoto 1996). However, for imperative 3D languages, there must be explicit ways such as using lines or arrows to represent the flow of control between program units (Stiles and Pontecorvo 1992, Geiger 1998).

3.2.2 Handling Branches and Loops

In structured program, branches and loops change the flow of execution. In visual programming languages in which some or a whole part of textual syntax is replaced by 3D icons (Stiles and Pontecorvo 1992), statements for branches and loops such as *if*, *switch*, and *while* statements are handled by representative 3D icons. In our research, not only the syntax of branches and loops are represented with 3D icons, but their semantics are also modeled using dynamic model types. Details of handling of the branches and loops will be discussed in Section 5.

3.2.3 Modularization and parameter passing

Program modularization makes large programs more manageable. With visual programming languages it is very convenient to represent the program modules and the relationship among them provided by visual scoping and explicit parameter passing. In our research, pre-defined 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 Blender interface in *RUBE*. Parameter passing is explicitly defined by connecting input ports to output ports between 3D icons in Blender interface.

3.2.4 Level of Details

Usually language constructors in 3D programming require more physical space than textual constructors do. 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 programmers or users. In the *RUBE* framework, for example, *while* statement can be modeled with more than one block or it can be simply hard coded into a single block.

3.2.5 Visual Execution

One of the advantages in visual programming comes from the visual execution of programs. Since a program is visually represented, it is easy to animate the execution of a program during runtime. The animation of a program execution can be done by adding some 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 the representative icons for that program module changes. When applied to a graphics program, the animation of the program model is visually represented while the result of the program execution, which is the construction of graphics objects, appears. When using Blender interface, a program model and the graphics objects resulted from the execution of the program model can coexist and be animated together in the same 3D space.

3.3 Metaphors in Software Visualization

In his celebrated work, *Rhetoric*, Aristotle said, “Ordinary words convey only what we know already; it is from a metaphor that we can best get hold of something fresh.” The word “metaphor” is derived from the Greek word “transfer.” Its primary function is the understanding of unknown thing from familiar concepts. Lakoff and Johnson (1980) 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. It provides insights to a novice user regarding the nature of a function or application. For an expert, dead metaphors turn to idioms with some communicative value (Pirhonen and Brewster 2001).

Computer programs can take advantage of metaphors, especially in their visual representations. Icons first introduced in programming when Smith constructed a special user interface for operations in programming language. He used icons to subsume the notions of variables, reference, data structure, and functions (Smith 1975). Inspired by PYGMALION, Xerox introduced the 8010 "Star" Information System with the desktop metaphor as a user interface in 1981 (Harslem and Nelson 1982). Ploix presented the use of metaphors such as a solar system, cities, and spiders for the visualization of Lisp programs using Zeugma, which is a programming environment for the construction, development, and experimentation of analogical representations of programs (Ploix 1996, Ploix2002).

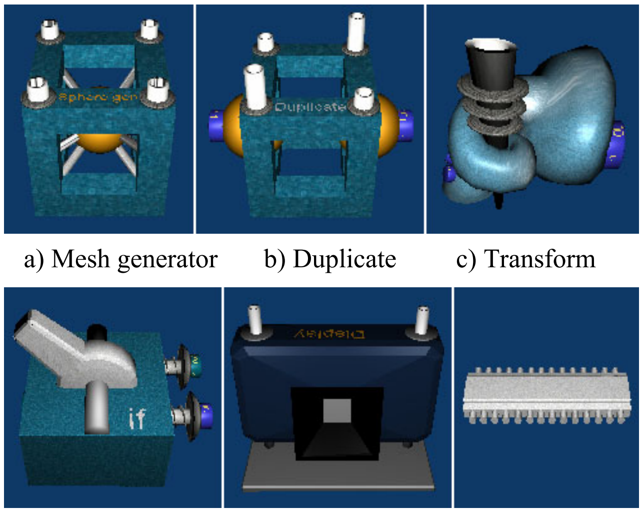
The conjunction of metaphors in visual programming could help learning programming. The use of metaphoric icons from which its functionality can be visually referred to would improve productivity and understandability of programming for novice programmers. Examples of metaphoric icons that might be suitable to be used in visual programming are: a warehouse metaphor for database applications where plenty of data is 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 chemistry metaphor for state-based applications where the current state of a system is determined based on the external or internal conditions.

While supporting user-created icons, *RUBE* provides different primitive and pre-defined 3D metaphoric icons for different themes. Primitive icons are cubes and spheres. Pre-defined themes include icons using the factory and the chemistry metaphors. In this paper, factory metaphors such as machines and conveyer belts are introduced in the representations of functions and traces in FBM. Figure 1 shows the parts of icons used in the factory theme.

The use of metaphors and customizable 3D components allow freedom to users in the model representation and stimulate user's creativity and interest. These also yield aesthetic aspects to be integrated with model visualization. Lavie and Tractinsky argued that the visual aesthetics of computer interface is a strong determinant of users' satisfaction and pleasure (Lavie and Tractinsky 2004). This is where our approach is differentiated from other works in visual programming that focus on fixed icons and presentation of the model structure.

4 IMPLEMENTATION

Previously, it was mentioned that the *RUBE* (Fishwick et al. 2003) framework provides the necessary environment for the construction and the execution of program models in our research. *RUBE* is a Web- and XML based modeling and simulation framework for geometry and dynamic models. *RUBE* includes a Python-based interface called Blender interface in which a user can define models and simulate them (Park and Fishwick 2004).



a) Mesh generator b) Duplicate c) Transform

d) Switch e) Display f) Trace

Figure 1: Factory Metaphor Icons

Using Blender interface, users can import the predefined or user-created Python modules into their program models. When a user locates and imports an icon into the program model using Blender Interface, a piece of Python code associated with it is also imported. The association between the Python module and its icon is set by placing both of them under the same directory in the RUBE file structure. Figure 2 shows the snapshot of the blender interface containing a program model that consists of User input, Intersect, and Display from the left-hand side. The trace icons represent the connectivity between the icons.

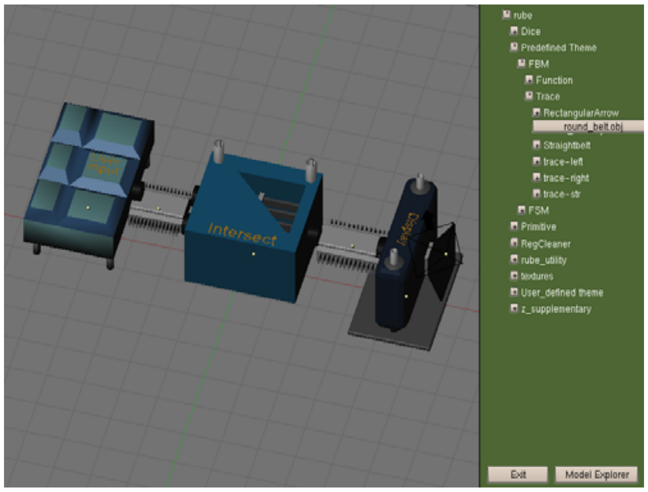


Figure 2: Snapshot of the Blender Interface with a Simple Model

Internal representation of program models is in XML. The components and structure of a program model defined in Blender 3D window is stored in a MXL file. MXL, which stands for Multimodel eXchange Language, is an application of XML developed by the modeling and simulation research group in University of Florida. 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 (Miller et al. 2001, Page 2000).

Once user-created model is stored in MXL, the MXL is translated into another XML modeling language, DXL–Dynamic eXchange Language. DXL is also developed by the modeling and simulation research group in University of Florida. While MXL maintains heterogeneous model types and uses different elements for different model types and model elements, DXL is a simple homogenous modeling language with blocks and connections (Lee and Fishwick 2002). The final simulation code in Python is generated from DXL. When DXL is translated into the actual Python simulation code, pieces of functional codes associated with each block in DXL are glued together. A simulation package called SimPack is also imported to provide various discrete-event simulation methods for model execution (Park and Fishwick 2002). Figure 3 shows the procedure of model translations in the *RUBE* framework.

5 GRAPHICAL PROGRAMMING USING PROGRAM MODELS

Computer graphics language is about creating and manipulating graphical objects. In the creation of computer graphics programs, graphics APIs such as OpenGL and Java3D provide a set of commands that allow the specification of geometric objects using the provided primitives, together with a set of commands that control how these objects are rendered. Another way to produce computer graphics is using authorizing tools. These are effective, and yet there remains the question of why we are not leveraging the power of computer graphics in the programming process itself.

In this section, constructing a graphics program that illustrates a blowing alley is discussed as a demonstrative example showing how 3D graphics is generated by modeling using a 3D APIs and dynamic modeling types in *RUBE* framework. This program model especially makes use of conditional branches in its program structure to control how many bowling pins to be added. There is more than one ways that we can model the conditional branches in using dynamic model types (Shim and Fishwick 2004).

Figure 4 shows FBM for the construction of a graphics program for a blowing alley with a floor, a ball, and pins. First, two execution threads are composed of simple operations such as texturing and transforming for the generation of a textured floor and a textured bowling ball, respectively. The circular connection of blocks in the third execution thread models the repeatable execution of *Duplicate* and *Transform* operations to produce of multiple pins. Once the control moves to the *If_Else* block, the execution of the thread continues or ceases depending on the condition specified in the *If_Else* block. In this example, when the execution is evaluated to be continued then *If_Else* block produces output to *Duplicate* block, otherwise it produces nothing. Figure 5 shows the program model built from Blender interface for the bowling alley example using the factory metaphor icons. A 3D graphics program is produced from the execution of the python code that is generated from this program model.

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. Using the *RUBE* framework it is possible to juxtapose the output graphics with the program model in the same 3D space. Figure 6 shows the snapshots of the inte grated graphics demonstrating the situation where the output graphics co-exists with its program model.

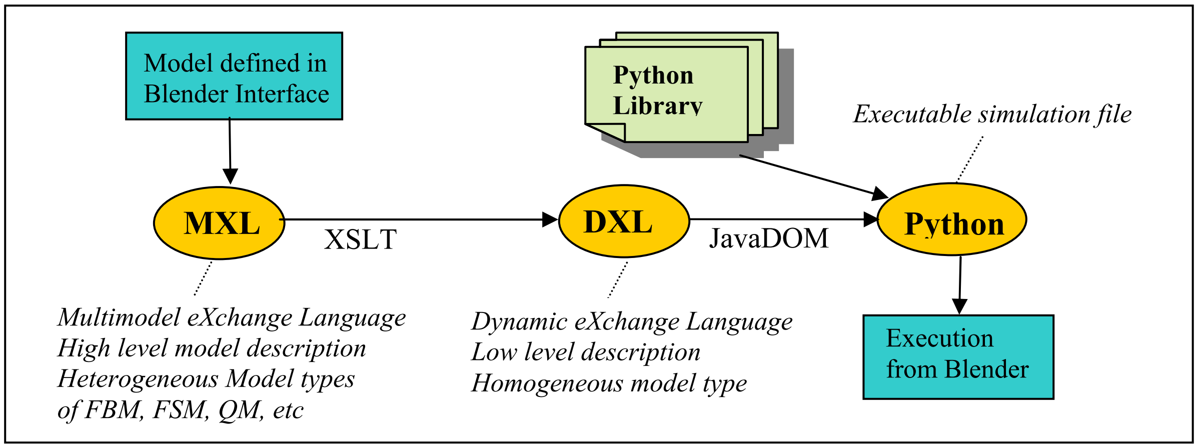

Model defined in Blender Interface
Python Library
Executable simulation file
MXL
XSLT
DXL
JavaDOM
Python
Multimodel eXchange Language
High level model description
Heterogeneous Model types
of FBM, FSM, QM, etc
Dynamic eXchange Language
Low level description
Homogeneous model type
Execution from Blender

Figure 3: Model Translations in *RUBE*

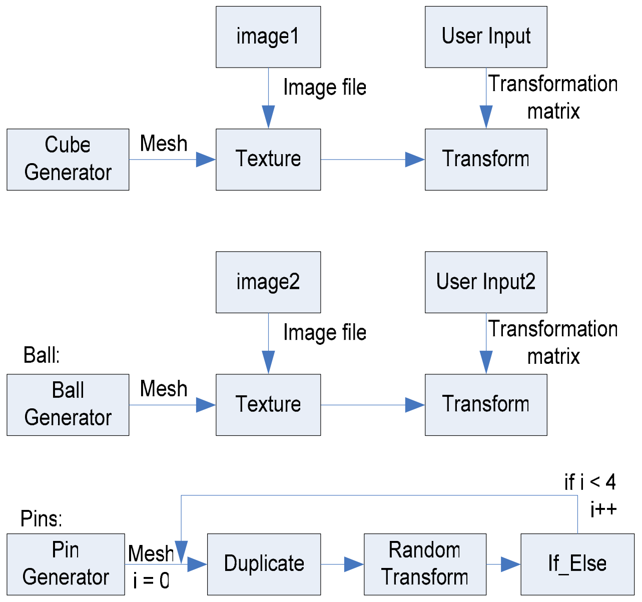


Figure 4: FBM for the Bowling Alley Generation

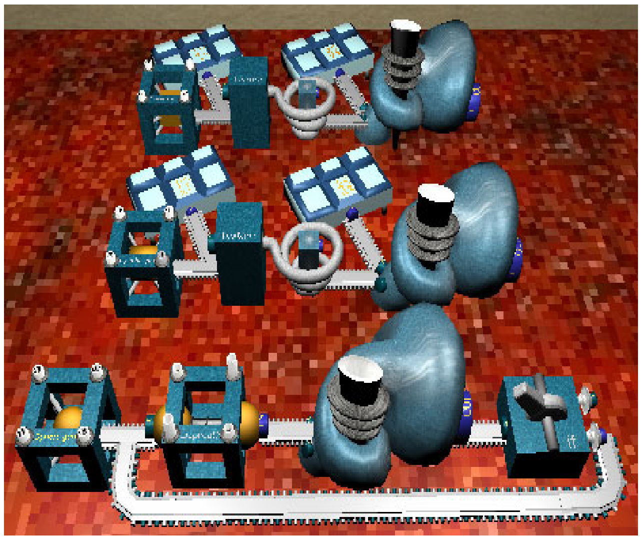


Figure 5: Program Model for the Bowling Alley Using a Factory Metaphor

By simply changing the parameters of *Texture* blocks and the iteration condition in *If_Else* block, we could get very different 3D graphics. Figures 7 and 8 are the 3D graphics produced from the execution of the program model in Figure 5 with different parameters for textures and iteration steps for the pin generation. Since the model components are distributed in a file structure, different 3D graphics 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 6, 7 and 8, Lightning effect such as ray-tracing and shading are added for better graphics.

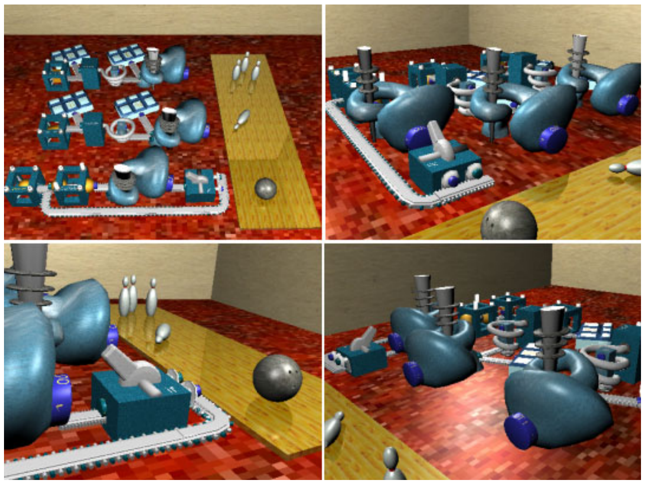


Figure 6: Program Model for the Bowling Alley Using a Factory Metaphor

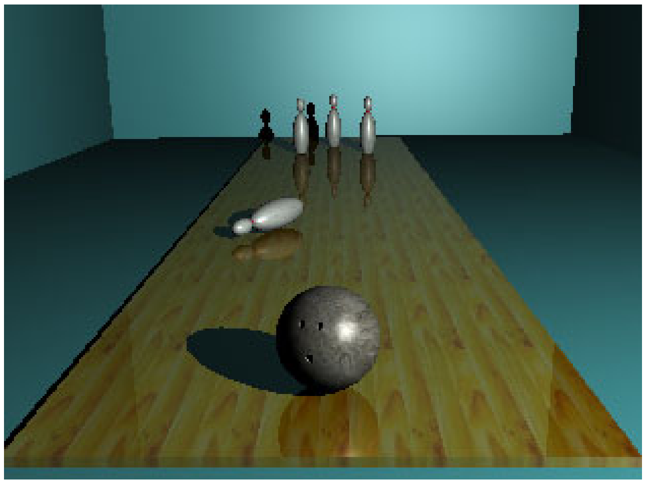


Figure 7: 3D Graphics Generated from the Program Model in Figure 5 with Four Iterations of the Pin Generation

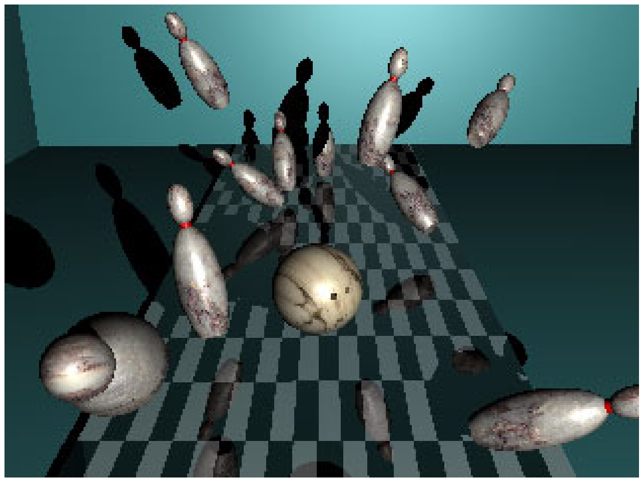


Figure 8: 3D Graphics Generated From the Program Model in Figure 5 with Ten Iterations of the Pin Generation

6 CONCLUSION AND FUTURE WORK

This research introduces a new approach to visual programming by integrating the principles and methodologies of modeling into programming. A visual programming environment, that facilitates a 3D API using dynamic model types, is introduced for the construction and execution of program models. 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 FBM and FSM. Instead of simple blocks and arrows, which are used in most of block diagrams, machines and conveyer belts in a factory are used as icons to represent program elements such as functions and parameters.

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

- Capturing dynamics of programs using simulation model types
- 3D program visualization
- Juxtaposing program models with program outputs in the same space
- Getting early feedback or debugging by running and modifying models
- Easy construction and execution of a program model in an integrated environment
- 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.

In Fall 2006, we will modify the computer graphics and simulation classes in University of Florida to use the approach and implementation to graphics programming in where students construct a 3D program and execute that program to generate 3D objects and animations as a part of their class project. We also plan to instrument assessment procedures about the students’ perceptions on how customized 3D visual programming affect their understanding and preferences regarding visual and interactive program representations.

In addition to that, as our future research, we will explore the possibilities of using other dynamic model types than the ones introduced in this paper to model other programming principles such as variables, inheritance and scoping. To handle variables in our research, the modification of MXL schema and translation engines will be required. We are also planning to increase the application of our research to more general programming domains.

ACKNOWLEDGMENTS

We would like to thank the National Science Foundation under grant EIA-0119532 and the Air Force Research Laboratory under grant F30602-01-1-05920119532 for support of this research.

REFERENCES

Banks, J., and J. S. Carson II. 1986. Introduction to Discrete-Event Simulation. *Proceedings of the 1986 Winter Simulation Conference*, ed. J.Wilson, J. Henriksen, and S. Roberts, 17 – 23.

Bloss, A. 1990. A Functional Approach to Simulation Programming. In *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, RP Sadowski, and. RE Nance, 214 – 219.

Fishwick, P. A. 1995. *Simulation Model Design and Execution: Building Digital Worlds*. Upper Saddle River: Prentice-Hall.

Fishwick, P. A., J. Lee, M. Park, and H. Shim. 2003. Rube: A Customized 2D and 3D Modeling Framework for Simulation. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, PJ Sanchez, D. Ferrin, and DJ Morrice, 755 – 762.

Geiger, C., W. Mueller, and W. Rosenbach. 1998. SAM-An Animated 3D Programming Language. In *Proceedings of 1998 IEEE Symposium on Visual Languages*, 228 – 235.

Haeberli, P. E. 1988. ConMan: A Visual Programming Language for Interactive Graphics. *Computer Graphics, SIGGRAPH 88 Conference Proceedings* 22 (4): 103 – 111.

Harslem, E., and L. E. Nelson. 1982. A Retrospective on the Development of Star. In *Proceedings of the Sixth International Conference on Software Engineering*.

Kahn, K. M., and V. A. Saraswat. 1990. Complete Visualizations of Concurrent Programs and Their Executions. In *Proceedings of the IEEE Visual Language Workshop*, 7 – 14.

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

Lavie, T., and N. Tractinsky. 2004. Assessing Dimensions of Perceived Visual Aesthetics of Web Sites. *International Journal of Human-Computer Studies* 60: 269 – 298.

Lee, G. A., G. J. Kim, and C. Park. 2002. Modeling Virtual Object Behavior within Virtual Environment. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology*, 41 – 48.

Lee, J., and P. A. Fishwick. 2002. A Dynamic Exchange Language Layer for Rube. In *Proceedings of Enabling Technology for Simulation Science,* Part of SPIE Aerosense ‘02 Conference, 359 – 366.

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

Nance, R. E. 1993. A History of Discrete Event Simulation Programming Languages. In *Proceedings of History of Programming Languages Conference* 28 (3): 149 – 175.

Oshiba, T., and J. Tanaka. 1999. 3D-PP: Three-Dimensional Visual Programming System. In *1999 IEEE Symposium on Visual Languages*, 13 – 16.

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

Park, M., and P. A. Fishwick. 2002. SimPackJ/S: A Web-Oriented Toolkit for Discrete Event Simulation. In *Proceedings of Enabling Technology for Simulation Science*, Part of SPIE Aerosense '02 Conference, 348 – 358.

Park, M., and P. A. Fishwick. 2004. An Integrated Environment Blending Dynamic and Geometry Models. *2004 AI, Simulation and Planning In High Autonomy Systems* 3397, 574 – 584.

Pirhonen, A., and S. Brewster. 2001. Metaphors and Imitation. *In the Workshop proceedings of PC-HCI*, 27 - 32.

Ploix, D. 1996. Building Program Metaphors. In *Proceedings of PPIG Workshop*, 125 – 129.

Ploix, D. 2002. Analogical Representation of Programs. In *Proceedings. First International Workshop on Visualizing Software for Understanding and Analysis*, 61 – 69.

Shim, H., and P. A. Fishwick. 2004. A Customizable Approach to Visual Programming using Dynamic Multimodeling. In *Proceedings of Enabling Technology for Simulation Science,* Part of SPIE Aerosense '04 Conference.

Smith, D. C. 1975. PYGMALION: A Creative Programming Environment. PhD dissertation, Stan-ford University.

Stiles, R., and M. Pontecorvo. 1992. Lingua Graphica: A Visual Language for Virtual Environments. In *IEEE Symposium on Visual Languages*, IEEE Computer Society Press, 255 – 227.

Sutherland, I. E. 1963. Sketchpad: A Man-Machine Graphics Communication System. In *Proceedings of the IFIP Spring Joint Conference*, 329 – 346.

Yamamoto, K. 1996. 3D-Visualan: A 3D Programming Language for 3D Applications. *Pacific Workshop on Distributed Multimedia Systems*, 199 – 206.

AUTHOR BIOGRAPHIES

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`.

PAUL A. FISHWICK is Professor of Computer and Information Science and Engineering at the University of Florida. He received the PhD in Computer and Information Science from the University of Pennsylvania in 1986, and has six years of industrial and government production and research experience (Newport News Shipbuilding and NASA Langley Research Center). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a Senior Member of the IEEE and a Fellow of the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which has served numerous subscribers. He has chaired several workshops and conferences in the area of computer simulation, including serving as General Chair of the 2000 Winter Simulation Conference. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the ACM Transactions on Modeling and Computer Simulation and the Transactions of the Society for Modeling and Simulation International. He has delivered 12 Keynote addresses at major conferences relating to simulation, and published over 150 technical publications, written one textbook, co-edited two Springer Verlag volumes in simulation, and published seven book chapters. He has recently edited Aesthetic Computing for MIT Press and is the Editor for the upcoming CRC Handbook on Dynamic Systems Modeling. His email and web addresses are `fishwick@cise.ufl.edu` and `http://www.cise.ufl.edu/~fishwick`.